No hay balas de plata. Esencia y accidentes de la ingeniería de software

Anthony Hall*
University of North Chapel Hill

Manejar constructos complejos es la esencia; tareas accidentales aparecen al representar estos constructos en algún lenguaje. El progreso en el pasado ha reducido las tareas accidentales de manera tal que el progreso ahora depende de resolver la esencia.

De todos los monstruos que llenan las pesadillas de nuestro folklore, ninguno aterroriza más que los hombres-lobo, porque se transforman de manera inesperada de algo familial a algo horroroso. Para ellos se buscan balas de plata que puedan destruirlos de forma mágica.

Los proyectos de software habituales, al menos tal como los ve un administrador no técnico, tiene algunas de estas características; generalmente se ve inocente y directo, pero puede transformarse en un monstruo de plazos no alcanzados, presupuestos sobrepasados, y productos con errores. Entonces se escuchan gritos desesperados pidiendo una bala de plata–algo que haga disminuir los costos del software tan rápidamente como bajan los costos del hardware.

Pero, si miramos con un horizonte de diez años, no vemos ninguna bala de plata. No existe ni un solo desarrollo,, ni en tecnología ni en administración, que por sí mismo prometa ni siquiera un orden de magnitud de mejoras en la productividad, confiabilidad, o simplicidad. En este artículo, trataré de mostrar por qué, examinando tanto la naturaleza de los problemas del software como las propiedades de las balas que han sido propuestas.

Sin embargo, escepticismo no es lo mismo que pesimismo. A pesar de que no vislumbramos grandes saltos cualitativos–y por cierto creo que esto es consistente con la naturaleza del software–existen en desarrollo muchas innovaciones promisorias. Un esfuerzo disciplinado y consistente para desarrollar, propagar y explotar estas innovaciones debería por cierto traer un orden de magnitud de mejora. No existe un camino real, pero este es el camino.

El primer paso hacia la cura de las enfermedades fue reemplazar las teorías de los demonios por la teoría de los gérmenes. Sólo ese cambio en sí, el comienzo de la esperanza, quita toda esperanza de soluciones mágicas. Esto enseñó a los trabajadores que el progreso se haría paso a paso, con gran esfuerzo, y que se debería tener una disciplina persistente con la limpieza. De la misma forma se encuentra hoy la ingeniería de software.

Es necesario que sea difícil? – Dificultades esenciales

No sólo no existe ninguna bala de plata a la vista, sino que la naturaleza del software en sí hace improbable que exista ninguna–ningún invento hará a la productividad, confiabilidad, y simplicidad lo que la electrónica, los transistores, y la integración a gran escala hicieron por el hardware de los computadores. No podremos entonces esperar mejorar las ganancias al doble cada dos años.

*Originalmente publicado en Information Processing 1986, ISBN No. 0444-7077-3, H. J. Kugler, Ed., Elsevia Science Publishers B.V. (North-holland) IFIP 1986. Traducido por María Cecilia Bastarrica en agosto de 2006.

Primeramente, debemos observar que la anomalía no es que el progreso del software es muy lento, sino que el progreso en hardware es muy rápido. Ninguna otra tecnología desde el inicio de la civilización ha tenido seis órdenes de magnitud de aumentos en sus ganancias en la relación precio-performance en 30 años. En ninguna otra tecnología se puede escoger tomar las ganancias o bien en mejoras en la performance o en reducir los costos. Estas ganancias provienen de la transformación de la industria de los computadores de una industria de ensamblaja a una industria de procesos.

En segundo término, para apreciar la tasa de progreso que se puede esperar en tecnologías de software, examinemos sus dificultades. De acuerdo con Aristóteles, las dividimos en esencia, o dificultades inherentes a la naturaleza del software, y accidentes, o sea esas dificultades que hoy enfrentamos pero que no son inherentes a la producción de software.

La esencia de una entidad de software es un constructo de conceptos interrelacionados: conjuntos de datos, relaciones entre estos datos, algoritmos, e invocaciones a funciones. Esta esencia es abstracta considerando que es la misma independientemente de su representación. Sin embargo, es muy precisa y ricamente detallada.

Creo que la parte más difícil de construir software es su especificación, diseño y prueba de estos constructos conceptuales, no la labor de representarlos y probar la fidelidad de dicha representación. Aún cometemos errores de sintaxis, con seguridad, pero son detalles comparados con los errores conceptuales en la mayor parte de los sistemas. Si esto es cierto, construir software siempre será difícil. Inherentemente no existe ninguna bala de plata.

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

Complejidad. Las entidades de software son más complejas para su tamaño quizás que ningún otro constructo humano porque no existen dos partes iguales (al menos sobre el nivel de sentencias). Si existen, hacemos que las partes similares sean una única rutina, cerrada o abierta. De esta forma, los sistemas de software difieren profundamente de los computadores, edificios o automóviles, donde existen abundantes elementos repetidos.

Los computadores digitales son en sí más complejos que la mayor parte de las cosas que las personas construyen: tienen un enorme número de posibles estados. Esto hace que concebir, describir y probarlas sea tan difícil. Los sistemas de software tienen órdenes de magnitud más estados que los computadores.

De forma similar, el escalamiento de un software no es meramente la repetición de los mismos elementos en tamaños más grandes; es también necesariamente un escalamiento en el número de elementos diferentes. En la mayor parte de los casos, los elementos interactuan entre sí de una forma no lineal, y la complejidad del todo aumenta más que linealmente.

La complejidad del software es una propiedad esencial, no accidental. Por lo tanto, las descripciones de un software que abstraen su complejidad, también suelen quitar su esencia. Durante tres siglos, las matemáticas y la física han hecho grandes esfuerzos para construir modelos simplificados de fenómenos complejos, derivando propiedades de estos modelos, y verificando estas propiedades a través de la experimentación. Este paradigma funciona porque las complejidades ignoradas en los modelos no eran las propiedades esenciales del fenómeno. Esto no funciona cuando las complejidades son la esencia.

Muchos de los problemas clásico del desarrollo de productos de software deriva de su complejidad esencial y su crecimiento no lineal con el tamaño. De la complejidad deriva la dificultad de comunicación entre los miembros del equipo lo cual lleva a errores en los productos, aumento en los costos, y retrasos en los plazos. De la complejidad también viene la dificultad de enumerar, y menos aún comprender, todos los estados posibles de un programa, y de eso proviene la no confiabilidad. De la complejidad de la funcionalidad proviene la dificultad de invocar una función, lo cual hace que los programas sean difíciles de usar. De la complejidad de las estructuras proviene la dificultad de extender programas a nuevas funcionalidades sin crear efectos laterales. También de la complejidad

de las estructuras proviene la dificultad de visualizar los estados que constituyen trampas a la seguridad.

No solamente problemas técnicos, sino también problemas administrativos provienen de la complejidad. Hace que la supervisión sea más difícil, y por lo tanto impide la integridad conceptual. Hace difícil encontrar y controlar todos los puntos potencialmente problemáticos. Crea una carga tremendamente pesada de aprendizaje y compresión que hace que la rotación de personal resulte un desastre.

Conformidad. Los ingenieros de software no están solos enfrentando la complejidad. La física trata con objetos tremendamente complejos aún a niveles "fundamentales". El físico trabaja a partir de la creenicia de que existen principios unificadores a ser descubiertos, ya sea en forma de quarks o la teoría de campos unificada. Eistein argumentaba que debía haber una forma simplificada de expresar la naturaleza de las cosas porque Dios no es caprichoso ni arbitrario.

No existe una fe tal en la ingeniería de software. Gran parte de la complejidad que se debe gestionar es arbitraria, forzada por las instituciones humanas y los sistemas a los que debe acomodarse sin ritmo ni razón. Estas interfaces varían, y de tiempo en tiempo, no porque sea necesario sino solamente porque fueron diseñadas por personas diferentes, en lugar de Dios.

En muchos casos, el software debe adecuarse porque es el que ha llegado último a la escena. En otros casos, debe adecuarse porque se percibe como el más adaptable. Pero en todos los casos, la mayor parte de la complejidad proviene del adaptarse a otras interfaces; esta complejidad no puede ser simplificada tan solo rediseñando el software por sí solo.

Cambio. El software está siempre bajo presión de cambio. Por supuesto que también lo están los edificios, automóviles y computadores. Pero los artículos manufacturados no se cambian frecuentemente después que han sido fabricados, sino que son cambiados en modelos posteriores, o se incorporan cambios esenciales a otras copias del mismo diseño esencial. Sacar ciertos automóviles del mercado es bastante infrecuente, así como también son raros los cambios en computadores ya vendidos. Ambos son mucho más infrecuentes que los cambios en software ya distribuido.

En parte esto se debe a que los sistemas de software son esencialmente su función en sí, y la función es la parte que más siente la presión del cambio. En parte también es porque el software puede modificarse más fácilmente—es un artefacto intelectual infinitamente maleable. Los edificios de hecho son modificados, pero los altos costos del cambio son entendidos por todos y sirven para desestimular dichos cambios.

Todo software exitoso eventualmente se cambia. Dos procesos ocurren aquí. Primero, cuando un producto de software resulta útil, las personas lo prueban en nuevos casos al borde o más allá del dominio original. Las presiones para extender la funcionalidad vienen directamente de los usuarios a quienes les sirve la funcionalidad básica y le inventan nuevos usos. En segundo término, el software exitoso sobrevive más allá del tiempo de vida útil del computador para el cual fue desarrollado. Tanto nuevos computadores como nuevos discos y nuevos dispositivos de despliegue aparecen, y el software debe adaptarse para sacar partido de estos nuevos elementos.

En resumen, los productos de software forman parte de una matriz cultural de aplicaciones, usuarios, leyes y computadores. Todos ellos cambian continuamente, y sus cambios inexorablemente fuerzan al software también a cambiar.

Invisibilidad. El software es invisible y no puede ser visualizado. Las abstracciones geométricas son herramientas poderosas. El plano de un edificio ayuda tanto al arquitecto como al cliente a evaluar los espacios, el flujo de personas, las vistas. Las contradicciones y las omisiones se hacen evidentes.

A pesar del progreso en el análisis de las estructuras del software, ellas aún son invisualizables y no permiten usar en ellas algunas de las más poderosas herramientas mentales. Los dibujos a escala de las partes mecánicas y las figuras de palitos de las moléculas, a pesar de ser abstracciones, sirven al mismo propósito. Una realidad geométrica se captura en una abstracción también geométrica.

La realidad del software no es inherentemente espacial. Por lo tanto, no tiene una representación geométrica directa de la forma en que existen mapas de la tierra, diagramas para los circuitos de silicio, y esquemas de conectividad de los computadores. Tan pronto como intentamos hacer diagramas de estructuras, incontramos que no existe uno solo sino una serie de grafos dirigidos superpuestos unos sobre los otros. Estos grafos pueden representar el flujo de control, el flujo de datos, los patrones de dependencia, secuencia temporal, y relaciones de nombres y espacio. Estos grafos generalmente no son siquiera planos, y menos aún jerárquicos. Por cierto, uno de las formas de establecer control conceptual sobre estas estructuras es forzarlos a ser jerárquicos.

A pesar de progreso en restringir y simplificar las estructuras del software, aún permanecen invisualizables, y por lo tanto no le permite a nuestra mente usar sus herramientas conceptuales más poderosas. Esta limitación no sólo impide el proceso de diseño dentro de nuestra mente, sino que disminuye la posibilidad de comunicación entre nuestras mentes.

Los Adelantos Pasados se deben a Dificultades Accidentales

Si examinamos los tres pasos del desarrollo de la tecnología del software que han sido más fructíferos en el pasado, descubrimos que cada uno de ellos atacó una dificultad mayor en la construcción del software, pero que esas dificultades han sido accidentales, y no esenciales. También podemos ver los límites naturales a la extrapolacin de cada uno de estos avances.

Lenguajes de Alto Nivel

Seguramente el mayor golpe de productividad, confiabilidad y simplicidad ha sido el uso progresivo de leguajes de programación de alto nivel. La mayor parte de los observadores afirma que los desarrollos han mejorado al menos en un factor de cinco en productividad, y con correspondientes mejoras en confiabilidad, simplicidad y comprensibilidad.

Qué logra un lenguaje de alto nivel? Libera al programa de la mayor parte de su complejidad accidental. Un programa abstracto consiste de constructos conceptuales: operaciones, tipos de datos, secuencias y comunicación. El programa de máquina concreto se refiere a bits, registros, condiciones, ramificaciones, canales, discos, y cosas por el estilo. Dado que el lenguaje de alto nivel incluye los constructos que podrían desearse en un programa abstracto y evitar los constructos de bajo nivel, elimina un nivel completo de complejidad que nunca fue inherente al programa en absoluto.

Lo máximo que puede hacer un lenguaje de alto nivel es proporcionar todos los constructos que el programador pueda imaginar en el programa abstracto. Ciertamente el nivel de nuestro pensamiento acerca de estructuras de datos, tipos de datos, y operaciones crece constantemente a una tasa que nunca decrece. Y los lenguajes de desarrollo también evolucionan para acercarse al nivel de sofisicación de sus usuarios.

Más aún, a cierto punto de la elaboración de un lenguaje de alto nivel se crea la necesidad de manejar una carga de conocimiento que incrementa y no reduce la tarea intelectual de los usuarios los cuales raramente usarán ciertos constructos esotéricos.

Tiempo Compartido

El tiempo compartido trajo una mejora sustancial en la productividad de los programadores y en la calidad de sus productos, aunque no tan grande como la de los lenguajes de alto nivel.

El tiempo compartido ataca una dificultad bastante diferente. Preserva la inmediatez y por lo tanto permite tener una vista global de la complejidad. La lenta desaparición de la computación por lotes significa que uno inevitablemente olvide la minucia de lo que se pensaba cuando se estaba programando, compilando y ejecutando. Esta interrupción es costosa desde el punto de vista del tiempo porque uno debe refrescar su propia memoria. El efecto más serio puede ser la pérdida de la noción de todo lo que realmente sucede en un sistema complejo.

Este cambio en la forma de ejecución, así como las complejidades del lenguaje de máquina, es una dificultad accidental más que esencial del proceso de software. Los límites de la contribuión potencial del tiempo compartido se derivan directamente. El efecto principal del tiempo compartido es el acortamiento del tiempo de respuesta. A medida que nos aproximamos a cero, en cierto punto pasará el umbral humano de lo detectable, o sea cerca de 100 milisegundos. Más allá de este umbral, no se esperan más beneficios.

Ambientes Integrados de Desarrollo de Programas

Unix e Interlisp han sido los primeros ambientes integrados de programación ampliamente usados, y parecen haber mejorado la productividad en algunos factores. Por qué?

Ellos atacan las dificultades accidentales que resultan de usar programas individuales en forma conjunta, brindando bibliotecas integradas, formatos de archivo unificados, y tubos y filtros. Como resultados, las estructuras conceptuales que en principio siempre podrían invocar, alimentar y usar otras, podrn de hecho hacerlo en la práctica.

Este avance estimuló como consecuencia el desarrollo de una serie de herramientas, dado que cada nueva herramienta puede aplicarse a cualquier programa que user los formatos estándar.

Debido a estos éxitos, los ambientes son el foco de gran parte de la investigación actual en ingeniería de software. Veremos sus potencialidades y limitaciones en la siguiente sección.

Esperanza de Plata

Consideremos ahora los desarrollos técnicos que más habitualmente se muestran como potenciales balas de plata. Qué problemas enfrentan–los problemas esenciales o las restantes dificultades accidentales? Ofrecen avances revolucionarios o incrementales?

Ada y otros lenguajes avanzados de alto nivel. Uno de estos desarrollos es Ada, un lenguaje de alto nivel de propósito general de los 1980's. Ada no solo refleja mejoras evolucionarias en los conceptos de lenguaje, sino que contiene características que promueven diseño moderno y modularización. Quizás la filosofía de Ada es más avanzada que el propio lenguaje Ada, debido a su filosofía de modularización, de tipos abstractos de datos, y de estructura jerárquica. Ada es un lenguaje excesivamente rico, un resultado natural del proceso de dejar los requisitos como parte de su diseño. Esto no es fatal, dado que se pueden usar subconjuntos del lenguaje como forma de aprendizaje, y futuros avances en el hardware se harán cargo del exceso de costos de compilación. Avanzar en la estructuración de los sistemas de software es ciertamente un muy buen uso para el aumento de MIPS que pueden comprarse con el mismo dinero. Los sistemas operativos, tan famosos en los 1960's por su

consumo de memoria y ciclos de procesador, han probado ser una excelente forma de usar algunos MIPS y bytes baratos de memoria como parte del progreso del hardware [9].

Sin embargo, Ada no será la bala de plata que mate al hombre lobo de la productividad del software. Después de todo sólo se trata de otro lenguaje de alto nivel, y el mayor beneficio de edstos lenguajes viene dado por la primera transición–de la complejidad accidental del lenguaje de máquina al planteo más abstracto paso a paso de las soluciones. Una vez quitados estos accidentes, los restantes son menores, y seguramente el beneficio de quitarlos será también menor.

Yo predigo que dentro de una década, cuando se haya establecido la efectividad de Ada, habrá hecho una diferencia sustancial, pero no debido a ninguna de las características del lenguaje en particular, ni siquiera por la combinación de todas ellas. Tampoco serán los ambientes de desarrollo en Ada la causa de las mejoras. La mayor contribución de Ada será que el haberse cambiado a Ada hará que los programadores adopten técnicas modernas de ingeniería de software.

Programación Orientada a Objetos. Muchos estudiantes tienen más esperanzas puestas en la programación orientada a objetos que en cualquier otra técnica [4]. Yo me encuentro entre ellos. Mark Sherman de Datmougth, hace notar en CSnet News, que se debe tener cuidado en distinguir dos ideas distintas que tienen el mismo nombre: tipos abstractos de datos y tipos jerárquicos. El concepto de tipo abstracto de datos es que el tipo de un objeto debe ser definido por un nombre, un conjunto propio de valores, y un conjunto propio de operaciones más que por una estructura de almacenamiento, la cual debería estar oculta. Algunos de estos ejemplos son los paquetes de Ada (con tipos privados) y los módulos de Modula.

Los tipos jerárquicos, como las clases de Simula'7, nos permiten definir interfaces generales que pueden ser refinadas mediante tipos subordinados. Los dos conceptos son ortogonales–se puede tener jerarquías sin encapsulamiento y encapsulamiento sin jerarquías. Ambos conceptos representan avances reales en el arte de construir software.

Cada uno de ellos quita una dificultad accidental del proceso, permitiendo al diseñador expresar la esencia del diseño sin tener que expresar una gran cantidad de material sintáctico que no agrega contenido a la información. Para ambos, tipos abstractos de datos y tipos jerárquicos, el resultado es quitar un tipo de dificultad accidental de alto nivel y permitir una expresión de alto nivel del diseño.

Sin embargo, dichos avances no pueden hacer más que quitar dificultades accidentales de la expresión del diseño. La complejidad del diseño en sí es esencial, y dichos enfoques no hacen nada al respecto. Se puede lograr un orden de magnitud de mejoras con programación orientada a objetos sólo si la especificación de tipos que aún tienen nuestros lenguajes de programación constituye nueve décimos del trabajo que implica el diseño de un producto de software. Yo lo dudo.

Inteligencia Artificial. Muchas personas esperan que los avances en inteligencia artificial brinden una revolución que traiga ganancias de varios órdenes de magnitud en la productividad y la calidad del software [6]. Yo no. Para ver por qué, debemos aclarar qué significa "inteligencia artificial".

D. L. Parnas ha clarificado el caos terminológico [10]:

> *Existen dos definiciones de IA bastante comunes hoy en día. IA-1: El uso de computadores para resolver problemas que previamente solamente podían resolverse aplicando inteligencia humana. IA-2: El uso de un conjunto específico de técnicas de programación conocidas como herísticas o programación basada en reglas. En este enfoque, se estudia qué heurísticas o reglas usan los expertos humanos para resolver problemas... El programa se diseña de modo que resuelva el problema de la misma forma que lo hacen los humanos...*

La primera definición tiene un significado resbaloso... Algo que puede satisfacer la definición IA-1 hoy, una vez que vemos cómo funciona el programa y comprendemos el problema, puede considerarse que ya no es IA... Desafortunadamente sólo puedo identificar un cuerpo de tecnología que es propio de este campo... La mayor parte del trabajo es específico al problema concreto, y se requiere cierta abstracción o creatividad para ver cómo puede ser transferido.

Coincido completamente con esta crítica. Las técnicas usadas en el reconocimiento de voz tienen poco en común con las que se usan en reconocimiento de imágenes, y ambas son diferentes de las técnicas usadas en sistemas expertos. He tenido dificultad en imaginar cómo por ejemplo el reconocimiento de imágenes puede tener un aporte sustancial en la práctica de la programación. Y lo mismo para reconocimiento de voz. La dificultad de construir software consiste en decidir qué se quiere decir, y no cómo decirlo. La facilidad para expresar sólo puede dar ganancias marginales.

Los sistemas expertos, IA-2, merecen un sección aparte.

Sistemas Expertos. La parte más avanzada de la inteligencia artifical y la más ampliamente aplicada es la tecnología para la construcción de sistemas expertos. Muchos cientistas del software trabajan duramente para aplicar esta tecnología en los ambientes de construcción de software [6, 10, 1]. A qué se refiere el concepto y cuáles son sus perspectivas?

Un sistema expertoo es un programa que contiene un motor de inferencia generalizada y una base de reglas, toma datos de entrada y suposiciones, explora las inferencias derivadas usando la base de reglas, obtiene conclusiones y consejos, y ofrece explicaciones de sus resultados mostrando al usuario la trazabilidad de su razonamiento. Los motores de inferencia típicamente pueden uar datos y reglas difusas o probabilísticas, además de lógica determinística.

Dichos sistemas ofrecen claras ventajas sobre los algoritmos programados que son diseñados para obtener soluciones a problemas particulares:

- La tecnología de los motores de inferencia se desarrolla de una forma independiente de las aplicaciones, y sólo luego se los aplica a variados usos. Se puede así justificar una gran inversión en el desarrollo de dichos motores de inferencia, y de hecho esta tecnología está bastante avanzada.
- Las partes cambiables particulares de la aplicación se codifican en la base de reglas de una manera uniforme, y existen herramientas para desarrollar, cambiar, probar y documentar esta base de reglas. Esto hace más sistemático el tratar gran parte de las complejidades de la aplicación en sí.

El poder de dichos sistemas no viene dado por los sofisticados mecanismos de inferencia, sino por la riqueza de las bases de conocimiento que reflejan el mundo real de una manera más ajustada. Creo que el avance más importante que ofrece esta tecnología radica en la separación de la complejidad de la aplicación de la programación propiamente tal.

Cómo puede usarse esta tecnología a la ingeniería de software? De muchas formas: dichos sistemas pueden sugerir reglas de inferencia, recomendar estrategias de testing, recordar errores frecuentes, y ofrecer ideas de posible optimización.

Consideremos un consejero de pruebas imaginario, por ejemplo. En su forma más rudimentaria, el sistema experto de diagnóstico es similar a una lista de chequeo de un piloto donde se enumeran sugerencias como posibles causas de dificultad. A medida que la base de reglas contiene más información acerca de la estructura del sistema, y a medida que la base de reglas tiene en cuenta de forma más sofisticada los síntomas de problema reportados, el consejero de pruebas se vuelve más particular con respecto a las hipótesis que genera y las pruebas

que recomienda. Dicho sistema experto difiere radicalmente de los convencionales donde la base de reglas se modulariza jerárquicamente de la misma forma que el producto de software de modo que cuando el producto se modifica modularmente, la base de reglas puede también modificarse modularmente.

El trabajo necesario para generar las reglas de diagnóstico es trabajo que debe hacerse de todas formas para generar un conjunto de casos de prueba para los módulos y el sistema. Si se hace de una forma apropiada y general, tanto con una estructura uniforme para las reglas como un buen motor de inferencia, podría reducir realmente el trabajo de generar casos de prueba en el desarrollo, así como también a lo largo de las modificaciones del sistema y sus correspondientes pruebas. De la misma forma se puede analizar la aplicabilidad de otros consejeros para otras partes de la tarea de construir software.

Existen muchas dificultades para el programador en la construcción de sistemas expertos consejeros útiles. Una parte crucial de nuestro escenario imaginario es el desarrollo de formas sencillas de pasar de la especificación de la estructura del programa a la generación automática o semiautomática de reglas de diagnóstico. Aún más difícil y más importante es la doble tarea de adquisición de conocimiento: encontrar expertos que puedan articular y analizar por qué hacen lo que hacen,y desarrollar técnicas eficientes para extraer lo que ellos saben y destilarlo en forma de bases de reglas. El prerrequisito esencial para construir un sistema experto es contar con un experto..

El mayor aporte de los sistemas expertos será probablemente el poner al servicio del programador inexperto la experiencia y la sabiduría acumulada por los mejores programadores. Este aporte no es menor. La brecha entre la mejor práctica en ingeniería de software y la práctica promedio es muy grande–quizás mayor que en ninguna otra rama de la ingeniería. Una herramienta que ayuda a la difusión de las buenas prácticas sería realmente importante.

Programación Automática. Por casi 40 años, la gente ha escrito acerca de "programación automática", o la generación del un programa para resolver un problema a partir de la especificación de dicho problema. Algunos hoy escriben como si esta tecnología fuese a brindar el próximo gran salto. Sin embargo, Parnas [10] cree que el término "programación automática" se usa por su glamourn, y no por su contenido semántico, y dice

> *En resumen, la programación automática ha sido siempre un eufemismo para la programación con un lenguaje de más alto nivel que el actualmente disponible para el programador.*

Y argumenta, en esencia, que en la mayor parte de los casos, es el método de solución y no el problema, lo que requiere ser especificado.

Se pueden encontrar excepciones. La técnica para construir generadores es muy potente y se usa con buenos resultados en los programas de ordenamiento. Algunos sistemas de integración de ecuaciones diferenciales también han permitido la especificación directa del problema, y los sistemas han definido los parámetros, escogiendolos desde una biblioteca de métodos de solución, y han generado los programas.

Estas aplicaciones tienen algunas propiedades muy convenientes:

- los problemas pueden ser caracterizados por un pequeño conjunto de parámetros
- existen muchos métodos conocidos para la solución que brindan las alternativas de la biblioteca
- un análisis extensivo ha llevado a reglas explícitas para seleccionar las técnicas de solución, dados los parámetros del problema

Es difícil ver cómo se generalizan esta técnicas a un universo mayor de sistemas de software, donde los casos con estas características tan primorosas son la excepción. Es difícil aún imaginar cómo podría ocurrir este gran avance en la generalización.

Programación Gráfica. Un tema que suele ser favorito como tesis doctoral en ingeniería de software es la programación gráfica o visula–la aplicación de gráfica computacional para el diseño de software [7, 11]. Alguna veces la promesa de dicho enfoque se establece como una analogía con el diseño de chips VLSI, donde la gráfica computacional juega un rol fundamental. Otras veces los teóricos justifican el enfoque considerando diagramas de flujo como el medio ideal de diseñar programas y dando facilidades poderosas para sus construcción.

Nada convincente ni excitante ha aparecido de estos esfuerzos. Y yo supongo que nada aparecerá tampoco.

Primeramente, como yo ya he sugerido [5], el diagrama de flujo es una abstracción muy pobre de la estructura del software. Por cierto, podría verse como un intento desesperado de encontrar un lenguaje de alto nivel de control para el computador propuesto por Burks, von Neumann y Goldstine. El diagrama de flujo ha sido elaborado tristemente en una forma de múltiples páginas de cajas conectadas que han probado ser un mal artefacto de base para el diseño.

Yo creo que el desarrollo del mercado masivo es la tendencia más profunda de la ingeniería de software en el largo plazo. El costo del software siempre ha radicado en el costo de su desarrollo y no en su replicación. El hecho de compartir estos costos aún entre un pequeño grupo de usuarios hace que estos disminuyan. Otra forma de verlo es considerar que el uso de n copias de un software multiplica por n la productividad efectiva de su desarrollador. Esto es una mejora de la productividad de la disciplina y de la nación.

El asunto esencial es la aplicabilidad. Acaso me sirve un software disponible para realizar mi tarea específica? Algo sorprendente ha sucedido en este aspecto. Durante las décadas del 50 y 60, todos los estudios concluyeron que los usuarios no estaban dispuestos a usar software ya desarrollado para apoyar sus procesos de remuneraciones, control de inventario, gestión de cuentas, y otros. Los requisitos eran demasiado específicos y las variaciones de caso a caso muy grandes. Durante los 80's, encontramos que los paquetes de software para apoyar estas funciones han proliferado. Qué ha cambiado?

Por cierto que no han sido los paquetes. Es posible que sean algo más generales y algo más configurables que en el pasado, pero no mucho. Tampoco han cambiado las aplicaciones. Es posible aún que las necesidades de los negocios y la ciencia de hoy sean más diversas y complicadas que hace 20 años.

El gran cambio ha sido el cambio en la relación de costos entre el software y el hardware. En 1960, el comprador de un computador de dos millones de dólares sentía que estaba en condiciones de pagar 250 mil dolares más por un sistema de remuneraciones a medida. Hoy, el comprador de un computador de 50 mil dólares no puede siquiera concebir pagar una cantidad similar por un sistema de remuneraciones a medida, y por lo tanto adapta su proceso de remuneraciones para adaptarlo al paquete disponible. Los computadores son ahora tan ubicuos, y también tan amados, que las adaptaciones son aceptadas como una necesidad ineludible.

Existen algunas excepciones. La generación de paquetes de software ha cambiado poco a través de los años: planillas de cálculo y sistemas administradores de bases de datos. Estas herramientas son poderosas, y han tardado algo en aparecer; pero tienen miles de usuarios y muchísimos usos, algunos de ellos bastante poco ortodoxos. Existen muchos libros acerca de como usar planillas de cálculo para abordar algunas tareas inesperadas. Aplicaciones que típicamente serían desarrolladas a medida en Cobol o RPG, ahora se hacen habitualmente con estas herramientas.

Muchos usuarios operan actualmente sus propios computadores todo el día sin siquiera programar una línea de código. Por cierto, muchos de estos usuarios no podrían programar sus propias máquinas, pero sin embargo son capaces de resolver problemas con ellos.

Creo que la estrategia de productividad más poderosa para el software de muchas organizaciones de hoy es equipar a los trabajadores no expertos en computación que se encargan de los problemas de la empresa con computadores personales y buen software de escritura, dibujo, almacenamiento, y planillas de cálculo para su libre uso. La misma estrategia, implementada con paquetes generales de matemáticas y estadísticas, junto con

algunas capacidades simples de programación, serán la realidad para cientos de científicos.

Refinamiento de requisitos y prototipación rápida. La parte más difícil de construir software es decidir qué construir. Ninguna otra parte del trabajo conceptual es tan difícil como establecer los requisitos técnicos necesarios, incluyendo todas las interfaces con las personas, las máquinas, y otros sistemas de software. Ninguna otra parte del trabajo desajusta el sistema resultante si se hace mal. Ninguna otra parte es más difícil de rectificar más adelante.

Por lo tanto, la función más importante que realiza el constructor del software para el cliente es la extracción y refinamiento iterativos de los requisitos del software. Porque la verdad es que el cliente no sabe lo que quiere. El cliente generalmente no sabe qué preguntas deberá contestar, y en general no ha pensado el problema con el nivel de detalle necesario para la especificación. Aún la respuesta más sencilla "Haga que el nuevo sistema de software trabaje exactamente como nuestro sistema manual de procesamiento de información" no es tan sencilla. Uno nunca quiere exactamente eso. Los sistemas de software complejos son cosas que actúan, que semueven y que trabajan. La dinámica de esa acción es difícil de imaginar. Por lo tanto para planear cualquier actividad de diseño de software es necesario iterar intensamente con el cliente y el diseñador como parte de la definición del sistema.

Yo iré un paso más allá y afirmaría que es relmente imposible que un cliente, aún trabajando con un ingeniero de software, pueda especificar completamente, precisamente y correctamente los requisitos exactos de un producto de software moderno antes de intentar varias versiones del producto.

Entonces, una de las formas más promisorias de la tecnología actual, y una que aborda la esencia y no el accidente del problema del software, es el desarrollo de enfoques y herramientas que permiten el desarrollo rápido de prototipos de los sistemas usando los prototipos como parte de la especificación iterativa de requisitos.

Un prototipo de un sistema de software es aquel que simula las interfaces importantes y ejecuta las funciones principales del sistema, pero no está sujeto a las misma restricciones de velocidad del hardware, tamaño o restricciones de costo. Los prototipos típicamente ejecutan las tareas habituales de la aplicación sin siquiera abordar los casos excepcionales, ni responder a datos de entrada incorrectos, o abortar de manera controlada. El propósito del prototipo es hacer evidente la estructura conceptual especificada para que el cliente pueda probar su consistencia y usabilidad.

Gran parte del proceso actual de obtención de software se basa en el supuesto de que se puede especificar satisfactoriamente el sistema por adelantado, obtener propuestas para su construcción, construirlo e instalarlo. Yo creo que este supuesto está equivocado fundamentalmente, y que existen una variedad de problemas en la obtención del software [2, 3, 8, 12].

Agradecimientos

Agradezco a Gordon Bell, Bruce Buchanan, Rick Hayes-Roth, Robert Patrick, y muy especialmente a David Parnas por sus ideas profundas y estimulantes, y a Rebekah Bierly por la producción técnica de este artículo.

Frederick P. Brooks es Kenan Professor de Ciencias de la Computación en University of North Carolina en Chapel Hill. Es más conocido como "el padre de la familia de computadores IBM sistema/360", y ha sido administrador del proyecto del hardware del sistema 360 y luego administrador de proyecto del software del sistema operativo 360.

En Chapel Hill, Brooks fundó el departamento de Ciencias de la Computación de la UNC y ha participado del establecimiento y la guía del Microelectronics Center of North Caroline, el Triangle Universities Computation

Center, y el North Carolina Educational Computingt Service. Ha recibido la National Medal of Technology, un Guggenheim fellowship, y los premios McDowell y Computer Pioneer de la Compueter Society de la IEEE.

Brooks recibió su doctorado en ciencias de la computación en Harvard, donde estudió con Howard Aiken.

Los lectores pueden escribirle a F. P. Brooks a la University of North Carolina, Dept. of Computer Science, Chapel Hill, NC 27514.

Para aniquilar al hombre lobo

Por qué una bala de plata? Magia, por supuesto. La plata se identifica con la luna y por lo tanto tiene propiedades mágicas. Una bala de plata ofrece la forma más rápida, poderosa y segura para aniquilar al rápido, poderoso e increiblemente peligroso hombre lobo. Y qué podría ser más natural que usar el metal de la luna para destruir a una criatura que se transforma bajo la luz de la luna llena?

La leyenda del hombre lobo es probablemente una de las más antiguas leyendas sobre monstruos. Herodoto, en el siglo quinto BC, nos da un primer informe escrito sobre los hombres lobe cuando menciona una tribu del norte del Mar Negro, llamada Neuri, quienes supuestamente se convierten en lobos por unos días cada año. Herodoto escribió que él mismo no lo creía.

Fuera de todo escepticismo, muchas personas creen que existe gente que se convierte en lobo o en otros animales. En la Europa medieval, algunas personas eran condenadas porque se pensaba que eran hombres lobo. En ese tiempo, no era necesario ser mordido por un hombre lobo para convertirse en uno más. Un pacto con el diablo, usar una posión especial, usar un cinturón especial, o estar embrujado por una bruja, todo podía convertir a una persona en hombre lobo. Sin embargo, los hombres lobo medievales podían ser heridos y morir con armas convencionales. El problema era superar sus encantos.

Entre los hombres lobo de la ficción y no de la leyenda. La primera película de hombres lobo, “El hombre lobo de Londres”, de 1935, creó un hombre lobo de dos piernas que se convertía en un monstruo con la luna llena. Se había convertido en hombre lobo cuando uno lo había mordido, y sólo podía matarse con una bala de plata. Suena familiar?

En realidad, le debemos gran parte de nuestras ideas sobre hombres lobo a Lon Chaney Jr. y su inolvidable performance en “Hombre lobo” de 1941. En películas posteriores casi no se muestra la mitologí de los hombres lobo mostrados allí. Pero esa película de todas formas se apartaba de la mitología original de los hombres lobo.

Creerían que antes de que la ficción se hiciera cargo de la leyenda las balas de plata no eran problema para los hombres lobo? Eran los vampiros los que les temían. Ciertamente, si nos basamos en las leyenda, su única salvación si lo ataca un hombre lobo es trepar a un fresno o correr hacia un campo sembrado con centeno. No muy fácil de encontrar en un ambiente urbano, y dificilmente reconocible por la media de la audiencia de las películas.

A qué se debe estar alerta? A las personas cuyas cejas se juntan, su dedo índice s más largo que el dedo mayor, y que les crece pelo en la palma de la mano. Tener dientes rojos o negros definitivamente son señales de problemas.

De todas formas tenga cuidado. Los mismos síntomas tienen las personas que sufren de hipertricosis (personas que nacen con pelo que cubre su cuerpo) o porfiria. En la porfiria, el cuerpo de la persona produce una toxina llamada porfirins. Como resultado, les molesta la luz, les crece pelo en el cuerpo, y los dientes se les pueden poner rojizos. Peor para la reputación de la víctima, su comportamiento crecientemente extraño hace que las demás personas noten sus síntomas con más suspicacia. Es muy probable que las personas que padecían esta enfermedad hayan contribuido involuntariamente con la leyenda, a pesar de que en los viejos tiempos no se los

acusaba de tendencias asesinas.

Referencias

[1] R. Baker. 15-year perspective on automatic programming. *IEEE Transactions on Software Engineering, Special Issue on Artificial intelligence and Software Engineering*, 11(11):1257–1267, November 1985.

[2] Defence Science Board. Report of the task force on military software, 1986.

[3] B. W. Boehm. A spiral model of software development and enhancement. Technical Report TRW 21-371-85, Space Park, Redondo Beach, CA 90278, 1985.

[4] G. Booch. *Object Oriented Design. Software Engineering with Ada.* Benjamin Cummings, Menlo Park, California, 1983.

[5] F. P. Brooks. *The Mythical Man-Month.* Addison-Wesley, Mass., New York, 1975.

[6] J. Mostow ed. *IEEE Transactions on Software Engineering, Special Issue on Artificial intelligence and Software Engineering*, 11(11), November 1985.

[7] R. B. Graphton and T. Ichikawa eds. *Computer, Special issue on visual programming*, 18(8), August 1985.

[8] H. D. Mills. Top-down programming in large systems. In R. Ruskin, editor, *Debugging Techniques in Large Systems.* Prentice-Hall, Englewood Cliffs, 1971.

[9] D. L. Parnas. Designing Software for Ease of Extension and Contraction. *IEEE Transactions on Software Engineering*, 5(2):128–138, March 1979.

[10] D. L. Parnas. Software aspects of strategic defense systems. *American Scientist*, November 1985.

[11] G. Reader. A survey of current fraphical programming techniques. *Computer, Special issue on visual programming*, 18(8):11–25, August 1985.

[12] H. Sackman, W. J. Erikson, and E. E. Grant. Exploratory experimental studies comparing online and offline programming performance. *CACM*, 11(1):3–11, January 1968.