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

De todos los monstruos ninguno aterroriza más que el hombre lobo, porque se transforma de manera inesperada de algo familiar a algo horroroso. Para ello 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 sencillo, 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, es decir, 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. Sin embargo, a pesar de que no vislumbramos grandes saltos cualitativos, 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.

**¿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.

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.

En segundo término, la esencia de una entidad de software es una construcción de conceptos interrelacionados: conjunto de datos, relaciones entre estos datos, algoritmos, e invocaciones a funciones. 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.

Las propiedades inherentes de esta esencia irreductible de los sistemas de software modernos son: complejidad, conformidad, variabilidad, e indivisibilidad.

**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.

Los ordenadores 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 probarlos sea tan difícil. Los sistemas de software tienen órdenes de magnitud más estados que las que tienen los ordenadores.

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.

Muchos de los problemas clásicos del desarrollo de productos de software derivan 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. 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 las estructuras proviene la dificultad de extender programas a nuevas funcionalidades sin crear efectos laterales.

No solamente problemas técnicos, sino también 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:** 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 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.

**Variabilidad:** el software está siempre bajo presión de cambio. 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).

Todo software exitoso eventualmente se cambia, debido a dos procesos. 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. Segundo, el software exitoso sobrevive más allá del tiempo de vida útil del computador para el cuál 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.

**Invisibilidad:** el software es invisible y no puede ser visualizado. A pesar del progreso en el análisis de las estructuras del software, ellas aún son no visibles y no permiten usar en ellas algunas de las más poderosas herramientas mentales.

La realidad del software no es inherentemente espacial. Por lo tanto, tan pronto como intentamos hacer diagramas de estructuras, encontramos que no existe uno solo sino una serie de grafos dirigidos superpuestos unos sobre los otros. Estos grafos generalmente no son siquiera planos, y menos aún jerárquicos que, por cierto, una de las formas de establecer control conceptual sobre estas estructuras es forzarlos a ser jerárquicos.

A pesar del progreso en restringir y simplificar las estructuras del software, aún permanecen no visuales, y por lo tanto no le permite a nuestra mente usar 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.

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

Un lenguaje de alto nivel logra liberar al programa de la mayor parte de su complejidad accidental. 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 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 sofisticación de los usuarios.

**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, ya que preserva la inmediatez y por lo tanto permite tener una vista global de la complejidad. El lento desarrollo del trabajo por lotes significaba que uno inevitablemente olvidaba la minucia de lo que se pensaba cuando se estaba programando y se preparaba para compilar y ejecutar.

Este cambio en la forma de ejecución, así como las complejidades del lenguaje máquina, es una dificultad accidental más que esencial del proceso de software. El efecto principal del tiempo compartido es el acortamiento del tiempo de respuesta.

**Ambientes integrados de desarrollo de programas:** los ambientes integrados de programación parecen haber mejorado la productividad en algunos factores. Atacan las dificultades accidentales que resultan de usar programas individuales en forma conjunta, brindando bibliotecas integradas, formatos de archivo unificados, y tubos y filtros.

Este avance estimuló como consecuencia el desarrollo de una serie de herramientas, dado que cada nueva herramienta puede aplicarse a cualquier programa que use los formatos estándar. Debido a estos éxitos, los ambientes integrados son el foco de gran parte de la investigación actual en ingeniería de software.

**Esperanzas de plata**

Consideremos ahora los desarrollos técnicos que más habitualmente se muestran como potenciales balas de plata.

**Ada y otros lenguajes avanzados de alto nivel:** Ada no sólo refleja mejoras evolucionarias en los conceptos de lenguaje, sino que contiene características que promueven diseño moderno y modularización. 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.

Sin embargo, Ada no será la bala de plata que mate al monstruo de la productividad del software. Después de todo sólo se trata de otro lenguaje de alto nivel, y el mayor beneficio de estos 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.

**Programación orientada a objetos:** muchos estudiantes tienen más esperanzas puestas en la programación orientada a objetos que en cualquier otra técnica.

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. Los tipos jerárquicos nos permiten definir interfaces generales que pueden ser refinadas mediante tipos subordinados.

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 de diseño. La complejidad del diseño en sí es esencial, y dichos enfoques no hacen nada al respecto.

**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.

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.

**Sistemas Expertos:** la parte más avanzada de la inteligencia artificial y la más ampliamente aplicada es la tecnología para la construcción de sistemas expertos.

Un sistema experto 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.

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.

Esta tecnología puede usarse en 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.

Existen muchas dificultades para el programador en la construcción de los asesores basados en sistemas expertos en el desarrollo de programas. Una parte crucial de nuestro escenario imaginario es el desarrollo de formas sencillas de pasar 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 los que hacen, y desarrollar técnicas eficientes para extraer lo que ellos saben y destilarlo en forma de bases de reglas. El pre-requisito esencial para construir un sistema experto es contar un con experto.

**Programación Automática:** durante casi 40 años, la gente ha estado anticipando y escribiendo sobre “programación automática”, o la generación de un programa para resolver un problema a partir de las especificaciones del problema.

En la mayoría de los casos las especificaciones que deben tenerse no son las del problema sino las del método para su solución. Hay excepciones. La técnica de construir generadores de código es muy potente, y se usa rutinariamente en programas para ordenar.

Estas aplicaciones tienen propiedades muy favorables: los problemas son claramente caracterizados por (relativamente) muy pocos parámetros. Hay muchos métodos conocidos de solución que proporcionan una librería de alternativas. Un análisis extensivo ha conducido a reglas explícitas para seleccionar técnicas de solución, a partir de parámetros dados por el problema.

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

**Programación gráfica:** un tema que suele ser favorito como tesis doctoral en ingeniería de software es la programación gráfica o visual. Algunas veces la promesa de dicho enfoque se establece como una analogía, dónde 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 su construcción.

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

Primeramente, el diagrama de flujo es una abstracción muy pobre de la estructura del software. Por cierto, podría verse como un intento desesperado de Burks, Von Neumann y Goldstine de encontrar un lenguaje de alto nivel de control para su propuesta de ordenador. 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. Los programadores dibujan los diagramas de flujo después, no antes de haber escrito los programas que describen.

Segundo, las pantallas de hoy son demasiadas pequeñas, en píxeles, para mostrar tanto el conjunto como la resolución de cualquier diagrama de software seriamente detallado. La tecnología de hardware tendrá que avanzar bastante más antes de que la amplitud de nuestros monitores sea suficiente para la tarea de diseñar software.

Aún más fundamental, es que el software sea muy difícil de visualizar. Si uno hace diagrama de flujo de control, el anidado de los ámbitos de las variables, las referencias cruzadas entre las variables, el flujo de datos, estructuras jerárquicas de datos o cualquier otra cosa, uno percibe sólo una dimensión del elefante de estructuras intricadamente interconectadas que es el software.

**Verificación del programa:** Gran parte del esfuerzo de la programación consiste en el testeo y la corrección de errores. No creo que podamos encontrar la magia de la productividad aquí. La verificación de programas es un concepto muy potente y será muy importante en cosas tales como kernels de sistemas operativos seguros. La tecnología no promete, sin embargo, ahorrar en trabajo. Las verificaciones cuestan tanto trabajo que sólo unos pocos programas son verificados.

La verificación del programa no implica programas a prueba de errores. No hay magia aquí. Incluso la verificación de programa perfecta sólo podría establecer que un programa cumple con sus especificaciones.

**Entornos y herramientas:** los editores y entorno de desarrollo inteligentes específicos a un lenguaje no se usan todavía ampliamente en la práctica, pero la mayoría de ellos prometen librarnos de los errores sintácticos y los errores semánticos simples.

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

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

**Estaciones de trabajo:** la composición y edición de programas y documentos es soportada totalmente por las máquinas de hoy en día. Seguro unas estaciones de trabajo más potentes serán bienvenidas. Pero no podemos esperar mágicos avances gracias a ellas.

**Ataques prometedores sobre la esencia conceptual**

Todos los ataques tecnológicos a los accidentes en el proceso del software están limitados fundamentalmente por la ecuación de la productividad. Si los componentes conceptuales de la tarea están ocupando ahora la mayoría del tiempo, entonces ninguna actividad sobre las partes necesarias del trabajo que consistan simplemente en la expresión de los conceptos dará grandes ganancias de productividad.

Deberíamos centrarnos en aquellos ataques sobre la esencia del problema del software, la formulación de estructuras conceptuales complejas.

**Comprar vs Construir:** la solución más radical para construir software es no hacerlo.

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

Cualquiera de estos productos es más barato de comprar que de hacer. Un software comprado cuesta lo mismo que un año de programador. Más aún, tales productos tienden a estar mucho mejor documentados y mejor mantenidos que el producto propio.

El desarrollo del mercado de masas es, creo, el mayor avance en la ingeniería del software. El coste del software ha sido siempre el de desarrollo, no el de copia. Al compartir el coste simplemente entre unos pocos usuarios reduce radicalmente el coste por usuario. El factor clave, por supuesto, es la aplicabilidad.

El gran cambio ha venido en la relación de costo hardware/software. En 1960, el comprador de una máquina de 2 millones de dólares sentía que podía afrontar otros 250 mil más por un programa de nóminas a la medida. Hoy, el comprador de una máquina para la oficina de 50 mil dólares no concibe afrontar un programa de nóminas a la medida, por lo que adapta su gestión de nóminas a los paquetes disponibles.

Ahora, muchos usuarios operan sus propios ordenadores con diversas aplicaciones sin haber escrito nunca un programa. De hecho, muchos de estos usuarios no pueden escribir nuevos programas para sus máquinas, pero sin embargo son adeptos a solucionar nuevos problemas con ellos.

**Refinamiento de requisitos y creación rápida de prototipos:** la parte más dura de hacer software es, precisamente, decidir qué hacer. Ninguna otra de las partes en que se divide este trabajo es tan difícil como establecer los requisitos técnicos detallados, incluyendo todas las interfaces con las personas, las máquinas y los otros sistemas de software. Ninguna otra parte del trabajo destroza tanto el resultado final si se hace mal.

Por tanto, la función más importante que realiza el diseñador de software para el cliente es la extracción iterativa y el refinamiento de los requisitos del producto. La verdad es que el cliente no sabe lo que quiere. El cliente normalmente no sabe que preguntas deben ser respondidas, y casi nunca ha pensado en los detalles del problema necesarios para la especificación. Por ello, al planificar cualquier actividad de diseño de software, es necesario permitir un trabajo iterativo extensivo entre el cliente y el diseñador como parte de la definición del sistema.

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

Desarrollo incremental: hacer crecer en vez de construir software. Deberíamos mirar a la naturaleza y estudiar la complejidad de las cosas vivas, en vez de las cosas muertas que hace el hombre. Aquí encontramos construcciones cuya complejidad nos estremece. El mismo cerebro es tan intrincado que no se le puede sacar un mapa, poderoso más allá de la imitación. El secreto es que creció, no se construyó.

Lo mismo debería ocurrir con nuestro software. El sistema debería primero conseguir que se ejecutara, incluso sin hacer nada útil, excepto llamar al conjunto apropiado de subprogramas que no hacen nada. Entonces, paso a paso, debería ser rellenado, desarrollando los subprogramas. La aproximación necesita el diseño top-down, lo que representa un crecimiento desde arriba hacia abajo del software. Cada función añadida y cada nueva provisión para datos o circunstancias más complejas crece orgánicamente a partir de lo ya existente.

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

Podemos conseguir buenos diseños siguiendo prácticas buenas en lugar de pobres. Las buenas prácticas de diseño pueden enseñarse. Los programadores están entre los más inteligentes de la población, por lo que pueden aprender buenas prácticas.

Los diseños excelentes proceden de diseñadores excelentes. La construcción de software es un proceso creativo. Una metodología sana puede potenciar y liberar la mente creativa.

Una mirada retrospectiva muestra que, aunque mucho software bueno y útil ha sido diseñado por comités y fabricado como parte de proyectos con varios apartados, aquellos sistemas de software que tienen fans excitados y apasionados son aquellos que son el producto de uno o sólo unos pocos diseñadores excelentes.

Por tanto, creo que el esfuerzo individual más importante que podemos hacer es desarrollar las formas de que surjan grandes diseñadores. La mayoría de las organizaciones realizan esfuerzos considerables buscando y cultivando a sus gestores; no conozco a ninguna que realice esfuerzos equivalentes para encontrar y desarrollar a los diseñadores excelentes de los que depende en última instancia la excelencia técnica de los productos.

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