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

by Frederick P. Brooks, Jr.

El artículo aborda la ilusión de encontrar soluciones mágicas o "balas de plata" en la Ingeniería del Software para resolver problemas como plazos incumplidos, objetivos fallados y productos defectuosos. Aunque hay innovaciones, mejorar la ingeniería del software requerirá esfuerzo consistente y disciplinado. El autor sostiene que no existe tal solución mágica.

El principio de la esperanza:
La naturaleza del software, en comparación con el hardware, impide la existencia de balas de plata. Se destaca que el progreso del hardware ha sido excepcionalmente rápido, pero el software avanza a un ritmo más lento debido a sus dificultades esenciales e inherentes. Aunque no hay soluciones mágicas, un esfuerzo consistente y disciplinado puede conducir a mejoras significativas en la Ingeniería del Software.

Dificultades:
La dificultad principal radica en la especificación, diseño y prueba de estos conceptos, más que en su representación. El autor subraya que el proceso de construcción de software siempre será desafiante, sin una solución mágica.

Esenciales	Accidentales
Dificultades inherentes a la naturaleza del software	Dificultades que se encuentran hoy en día pero que no son inherentes al software.

Propiedades Inherentes al Software	
Complejidad	La complejidad del software supera a la de cualquier otra construcción humana (computadoras, edificios o autos) debido a la falta de **elementos repetidos**, (no existen dos iguales, y si los hay, se convierten en subrutinas) y la **no linealidad** en el aumento de complejidad con el tamaño. La complejidad del software es una propiedad **esencial** y no accidental. Eliminar su complejidad significa **eliminar su esencia**. Deriva en dificultades técnicas y también de gestión: Hay problemas de comunicación; la complicación de enumerar y de comprender, todos los posibles estados del programa conlleva a la falta de fiabilidad; dificultad en agregar nuevas funciones sin crear efectos laterales; y los estados no previstos que se convierten en agujeros en la seguridad.
Conformidad	A diferencia de otras disciplinas como la física, los **principios fundamentales** o **generales** (como leyes del movimiento o las leyes de termodinámica) proporcionan una base unificadora que se aplica de manera consistente en diversas situaciones.

	Sin embargo, en el campo del **software**, cada proyecto tiene sus requisitos, interfaces específicas y tecnologías, que hacen **imposible** la **aplicación** de **principios generales**. Por lo tanto, los ingenieros de software **no pueden depender** completamente de **principios generales** que **abarquen todas** las **situaciones**.
Variabilidad	El software vive dentro de un entorno que cambia continuamente, y esos cambios lo fuerzan para que se produzcan cambios en él. Es una característica propia del software exitoso **adaptarse a nuevas funciones** y **sobrevivir más** allá de la **vida útil del hardware** para el que fue diseñado.
Invisible	El software **no tiene representaciones espaciales** ni una **forma visual concreta**, debido a su naturaleza abstracta y no física. A diferencia de los objetos que nos rodean que se pueden representar mediante planos, esquemas, etc. Los **diagramas** de las **estructuras** de software, están compuestas por varios **gráficos superpuestos** en diferentes direcciones, que dificultan el pensamiento y además la comunicación sobre el diseño del software.

Algunos avances solucionaron dificultades, pero claro está, que esas dificultades son “accidentales” y no “esenciales”

Avances que solucionan dificultades accidentales			
Avances	**Dificultad Accidental**	**Solución**	**Limitaciones**
Lenguajes de alto nivel	Programar en lenguajes de bajo nivel requiere lidiar con detalles específicos del hardware o la implementación (bits, registros y condiciones).	Aumento en productividad, fiabilidad y simplicidad. Los programadores pueden escribir código de una manera más abstracta y comprensible (abstractizan detalles de bajo nivel)	La introducción de características muy especializadas pueden aumentar la carga intelectual del usuario en lugar de reducirla.
Tiempo compartido	Trabajar en entornos por lotes con largos tiempos de espera entre la escritura del código y su ejecución, dificulta la retención de detalles y la visión integral del desarrollo.	Aumento en productividad y calidad. Ofrece inmediatez: significa que los programadores pueden obtener una visión completa y actualizada del sistema en el que están trabajando.	Beneficios limitados cuando el tiempo de respuesta se aproxima a la percepción humana (alrededor de 100 milisegundos).
Entornos de desarrollo unificados.	Usar herramientas y formatos dispares entre programas individuales	Mejora en la productividad EDU integra herramientas, librerías y formatos	Puede volverse complejo a medida que se agregan más funciones; dependencia de

	puede generar complejidades al integrarlos.	estandarizados para facilitar la colaboración y reducir la complejidad al trabajar con varios programas.	estándares y compatibilidad entre herramientas.

Esperanzas de descubrir la plata:

Los desarrollos técnicos ¿Encaran problemas de qué tipo, esenciales o los accidentales? ¿Ofrecen avances revolucionarios o sólo mejoras incrementales?

Ada y otros avances en lenguajes de alto nivel:

El texto destaca Ada, un lenguaje de alto nivel de propósito general que presenta mejoras evolutivas en conceptos de lenguaje: enfatiza la modularidad, tipos de datos abstractos y estructuras jerárquicas.
A pesar de ser prolijo, su filosofía se considera un avance mayor que el propio lenguaje.

En resumen, Ada se valora más por influir en las prácticas de diseño de software que por ser la solución definitiva en sí misma. No ha demostrado ser la bala de plata que acabe con el monstruo de la productividad del software.

Avances que solucionan dificultades accidentales	
Programación orientada a objetos:	Abstracción de tipos de datos y jerarquías para facilitar el diseño y la ocultación de detalles internos. Los tipos jerárquicos permiten definir interfaces generales que se pueden refinar con tipos subordinados. permitir un mayor nivel de expresión en el diseño. Puede haber desafíos en la transición y comprensión para aquellos no familiarizados con el paradigma orientado a objetos.
Inteligencia Artificial	mejora de la toma de decisiones en el desarrollo de software. Uso de técnicas de IA, como sistemas basados en reglas y heurísticas, para resolver problemas específicos. El autor sostiene que el desafío fundamental no está en expresar o implementar soluciones técnicas (como reconocimiento de voz), sino en tomar decisiones sobre qué soluciones o funcionalidades son realmente necesarias y efectivas en el software. En otras palabras, considera que la parte más difícil no es la ejecución técnica, sino la toma de decisiones estratégicas y conceptuales durante el proceso de desarrollo de software.
Sistemas expertos	Simplificación y difusión del conocimiento experto en desarrollo de software. Utilización de motores de inferencia y bases de reglas para ofrecer consejos y conclusiones en el desarrollo.la contribución más significativa puede ser la difusión de la experiencia de programadores expertos a programadores menos experimentados, mejorando así las prácticas de diseño. Limitado por la cantidad y calidad de reglas y experiencia programática incorporada en el sistema.

Programación "Automática	Generación de programas a partir de especificaciones de problemas. Enfoque en la generación de programas a partir de especificaciones, especialmente en casos donde las soluciones son más predecibles. Generalización limitada de estas técnicas al software ordinario, donde la diversidad de problemas y propiedades favorables es menos común.
Programación gráfica	Mejora de la visualización y diseño a través de representaciones gráficas. Uso de gráficos de computadora para representar la estructura del software y facilitar el diseño. Limitaciones en la capacidad para visualizar la complejidad del software, especialmente en pantallas pequeñas. Se cuestiona la utilidad de los flujogramas como una abstracción efectiva de la estructura del software.La analogía con el diseño de chips puede no ser aplicable.
Verificación del programa	Testeo y corrección de errores en la fase de diseño del sistema es potente y esencial en casos como kernels de sistemas operativos seguros no implica programas libres de errores y que, incluso con una verificación perfecta, la tarea más difícil sigue siendo la obtención de especificaciones completas y consistentes.
Entornos y herramientas	seguimiento de detalles y productividad errores sintácticos y semánticos simples Edición inteligente específica de lenguaje Ganancias marginales
Estaciones de trabajo	Reducción del tiempo necesario para desarrollo Mejora en composición y edición de programas aumenta la potencia y memoria de las estaciones de trabajo individuales.pueden mejorar la velocidad de compilación

Ataques prometedores sobre la Esencia:
En este fragmento, se exploran diversas estrategias para mejorar la productividad y calidad en el desarrollo de software, destacando la importancia de atacar la esencia del problema. Se mencionan tres enfoques prometedores:

1. Comprar vs Construir:
- Se plantea la idea radical de no construir software y, en cambio, adquirir productos ya existentes en el mercado.
- Se destaca la expansión del mercado de software y la conveniencia de comprar productos más baratos y bien documentados en lugar de desarrollarlos internamente.
- Se enfatiza que el desarrollo masivo del mercado ha sido el mayor avance en ingeniería de software, reduciendo significativamente el costo por usuario.

2. Refinamiento de Requisitos y Prototipado Rápido:
- Se resalta la dificultad de establecer requisitos precisos para un sistema de software sin un proceso iterativo y de refinamiento.

- La importancia del prototipado rápido se destaca como una herramienta esencial para hacer tangibles las estructuras conceptuales y permitir que los clientes verifiquen la consistencia y usabilidad del sistema propuesto.

3. Desarrollo Incremental:

- Se propone una aproximación radicalmente diferente: hacer crecer el software de manera incremental en lugar de construirlo.
- Se argumenta que las estructuras conceptuales actuales son demasiado complejas para especificarse con precisión desde el principio y que el desarrollo incremental permite un crecimiento orgánico desde arriba hacia abajo.
- Se menciona la efectividad de esta estrategia en términos de resultados impresionantes y un aumento en la moral de los equipos de desarrollo.

En resumen, el autor aboga por **enfoques pragmáticos y radicales** que ataquen directamente los desafíos fundamentales del desarrollo de software, proponiendo soluciones como la **adquisición** en lugar de la construcción, el **prototipado rápido** y el **desarrollo incremental**.

Grandes Diseñadores

En este fragmento, se subraya la importancia del **diseño** de calidad en el desarrollo de software, enfocándose en la distinción entre **buenos y excelentes diseños**. Se argumenta que, aunque las buenas prácticas de diseño pueden enseñarse, alcanzar diseños de calidad excepcional **depende** en gran medida de la **creatividad individual de los diseñadores**.

Ejemplos de sistemas exitosos son: Unix y Pascal, que fueron productos de diseñadores excepcionales, a diferencia de proyectos menos emocionantes creados por comités.

El desafío radica en el desarrollo activo de diseñadores excepcionales, y se sugiere que las organizaciones deben reconocer y recompensar a estos diseñadores de manera equiparable a los gestores. Además, se proponen estrategias específicas, como identificar talento temprano, asignar mentores y proporcionar oportunidades de aprendizaje y colaboración para fomentar el crecimiento de diseñadores excepcionales en la industria del software.