**U1 - Resumen No Silver Bullet**

Los proyectos de software tienen muchos problemas, como retrasos en el desarrollo, presupuestos sobrepasados y productos de baja calidad. Y se busca constantemente algo que haga que los costos de desarrollo disminuyan, y mejore así la productividad, la confiabilidad y la simplicidad; sin embargo no hay ninguna disciplina técnica o de gestión que lo pueda lograr.

**Principales dificultades**

No es que el progreso del software es lento, sino que el progreso del hardware de computadoras es muy rápido (debido a que pasó de una industria de ensamble a una industria de proceso continuo)

Las dificultades se pueden dividir en dos: las dificultades inherentes de la naturaleza del software y los accidentes (que afectan al software pero no son inherentes del mismo). La parte difícil de construir software es el construir y diseñar el concepto del mismo, no el armarlo.

**Propiedades inherentes del software:**

**Complejidad**: Las entidades de software son complejas por naturaleza. Además escalar una entidad de software no es repetir elementos pero más grandes, sino que implica un incremento en los distintos elementos que lo conforman. A la hora de escalar, el tamaño del software incrementa de manera exponencial, no lineal.

Es difícil describir el software abstrayendo la complejidad del mismo, ya que dicha complejidad es inherente al mismo, no como el caso de los problemas matemáticos (ej: fórmulas matemáticas que solo funcionan en el vacío, como el vacío/presión del aire no forma parte del fenómeno, se puede abstraer). Ejemplos de problemas que trae la complejidad:

* Complejidad - Dificultad para comunicarse con los miembros del equipo.
* Complejidad - Dificultad para entender y enumerar los estados del programa.
* Complejidad de estructura - Dificultad para extender el programa con nuevas funciones sin crear efectos secundarios.
* Complejidad - Dificultad en la gestión, es difícil de supervisar, mantener la integridad.

**Conformidad**: Mucha de la complejidad que enfrenta el software es debido a la conformidad. Es una complejidad arbitraria que surge porque en muchos casos, el software se debe conformar con otras interfaces, es decir, se debe diseñar el software para adaptarse a dichas interfaces.

**Posibilidad de cambio:** La entidad de software constantemente está bajo presión para cambiar. El software está embebido en numerosas aplicaciones, usuarios, leyes y maquinaria, y cómo estos cambian continuamente, fuerzan los cambios en el producto de software. Además, como el software es algo intangible, completamente armado desde una idea, motiva a los clientes a querer cambiarlo. Muchas veces el cambio viene porque si un producto de software es útil, se empieza a presionar para extender su funcionalidad. Otras veces, como la vida del software es más larga que otras tecnologías, el software debe poder conformarse a los mismos (nuevos discos, nuevas impresoras, etc).

**Invisibilidad**: El software es invisible e intangible. El software no se puede representar en la realidad de ninguna forma. Cuando tratamos de diagramar la estructura del software, no se puede hacer en un solo gráfico, sino que necesitamos de muchos grafos dirigidos uno arriba del otro. A pesar del esfuerzo en restringir y simplificar las estructuras del software, aún permanecen inherentemente invisibles. Esta falla provoca que sea muy complejo realizar el proceso de diseño con múltiples mentes trabajando al mismo tiempo.

**Resolviendo las dificultades accidentales**

Si examinamos el pasado del desarrollo de software, nos encontramos que hubo avances respeto a las dificultades de hacer software, pero todas estas soluciones trataron dificultades accidentales, y no las inherentes del software:

**Lenguajes de Alto Nivel:** Atacaron la complejidad que proviene de escribir código en lenguajes de bajo nivel, y aumentaron la productividad. Pero el nivel del lenguaje no es inherente del software, por lo que es una dificultad accidental. Además, los lenguajes de alto nivel pueden disminuir la complejidad hasta cierto punto, luego, no pueden hacer mucho más.

**Time-Sharing**: Es otra dificultad accidental, en lugar de una inherente del software, me permite lograr un tiempo de respuesta menor, pero una vez reducido a 100ms, cualquier mejora no nos brinda beneficios, porque el ser humano no puede darse cuenta.

**Entornos de Programación Unificados**: Si bien proveen herramientas útiles para el desarrollo, librerías unificadas, formatos de archivo unificados, entre otros, no pueden hacer mucho más.

**¿Esperanzas?**

Hay muchas técnicas y tecnologías que se tratan como posibles soluciones al problema de la productividad, sin embargo, todas siguen tratando problemas accidentales y no inherentes. Todas tienen potencial de mejorar la productividad, pero tienen un límite claro, más allá del cual, no nos brindarán beneficios. Ejemplo:

* Avances en los lenguajes de alto nivel.
* Programación Orientada a Objetos
* Inteligencia Artificial
* Sistemas Expertos
* Programación Automática.
* Programación Gráfica.
* Verificación de Programas
* Entornos y herramientas
* PC

**Ataques a la dificultad conceptual**

La mayoría de ataques al problema son tecnológicos, y siempre tratan de aumentar la productividad, pero no se enfocan en la parte conceptual del software, por ello es que no tienen éxito. Sin embargo, hay algunas mejoras que prometen resolver la complejidad producida por las estructuras conceptuales:

**Comprar vs Hacer**: Cada día hay más softwares enlatados que me brindan ciertos aspectos de funcionalidad. Surge un mercado masivo de software, que me permite comprar a poco precio un producto genérico, pero que me permita realizar la tarea. Inclusive por el costo de $100.000 usd, puedo comprar un programa de software, y me costará lo mismo que pagarle a un solo programador durante un año, y el software lo voy a tener de manera inmediata. Sin embargo, esto no es de utilidad a empresas que necesitan programas especializados, el problema es la aplicabilidad. Con el abaratamiento de las computadoras, y el surgimiento de herramientas como Excel, las empresas cada vez pretenden gastar menos en el software, no se le puede pedir una inversión al nivel de millones para tener su propio software para pagar a los empleados, cuando por unos cuantos miles de dólares pueden tener lo mismo y de manera inmediata.

**El software debe crecer incrementalmente, no se debe construi**r: El sistema primero debe funcionar, andar, aunque no tenga nada de funcionalidad de alto nivel implementada. Permite una mejor trazabilidad, tener prototipos en etapas tempranas. Cada funcionalidad o estructuras de datos complejas, crece de manera orgánica en lo que ya poseíamos anteriormente.

**Grandes diseñadores**: La construcción de software es un proceso creativo, y los grandes diseños provienen de grandes diseñadores. Si bien es importante tener un buen método o proceso de diseño, con esto no es suficiente. Los grandes diseñadores y grandes administradores son bastante raros de encontrar, y las organizaciones realizan muchos esfuerzos para encontrarlos.