Manifiesto y Paper

▼ No Silver Bullet

▼ Introducción

▼ Cuál es la analogía que plantea el autor?

Frederick Brooks utiliza la analogía de la "bala de plata" (silver bullet) proveniente del folclore sobre los hombres lobo. En esas historias, solo una bala de plata podía matar a un hombre lobo, lo que simboliza una solución mágica y única para un problema complicado.

Un proyecto de software es como un hombre lobo, puede pasar de lo familiar o de lo sencillo hacia un monstruo de forma repentina y para un hombre lobo buscamos balas de plata, pero para un proyecto de software no existe una solución rápida y única que pueda resolver todos los problemas

▼ Cuáles son las 2 complejidades que distingue el autor?

Brooks distingue entre dos tipos de complejidad en la ingeniería de software:

1. **Esencial**: Se refiere a la dificultad intrínseca del problema que el software busca resolver, como los requisitos cambiantes y la necesidad de precisión.
2. **Accidental**: Son las dificultades derivadas de las herramientas y técnicas utilizadas para implementar el software.

▼ Qué representa el monstruo (hombre lobo) en un proyecto de software?

Representa los desafíos fundamentales y complejos del desarrollo de software, particularmente los que son de naturaleza **esencial**

Estos desafíos incluyen la complejidad inherente del problema que el software busca resolver, los requisitos cambiantes, la necesidad de precisión en el diseño y la dificultad para captar y modelar la realidad en código.

▼ Qué pasa en el software a diferencia del hardware?

El software es más complejo, ningún invento de los que mejoraron la productividad, fiabilidad y simplicidad en el hardware, como la electrónica, los transistores etc, harán lo mismo por el software.

No podemos esperar a que los costos del sw caigan tan rápidamente como los del hw → para el HW sí hubo balas de plata

▼ Dificultades esenciales

▼ Cuál es la esencia de una entidad software?

La esencia de una entidad de software, según Frederick Brooks en *"No Silver Bullet"*, se refiere a su **naturaleza intrínseca y fundamental**, es decir, las características que definen su propósito y comportamiento.

Brooks sostiene que esta esencia es la parte más difícil y compleja del desarrollo de software y que no puede eliminarse mediante mejoras tecnológicas o metodológicas

▼ Según el autor cuál es la parte más dura de construir SW?

La especificación, el diseño y prueba del concepto construido → no el trabajo de representarlo y comprobar la fidelidad de la representación

"Todavía tendremos errores de sintaxis, evidentemente; pero eso es trivial si lo comparamos con los errores conceptuales en la mayoría de los sistemas"

Básicamente la parte dura es la complejidad esencial, no la accidental (técnica)

▼ Cuáles son las dificultades esenciales en el desarrollo de SW?

▼ **Complejidad**

▼ La complejidad es una propiedad esencial o accidental?

La complejidad del software es una propiedad esencial, no accidental

▼ Por qué decimos que son complejas las entidades de SW?

- Por su tamaño
- Porque no hay 2 partes iguales. No hay elementos repetidos, a diferencia de otros sistemas.

- Tiene un gran número de estados e interacciones → lo que hace que su concepción, descripción, y pruebas sean muy duras
- Ante un incremento → el número de interacciones entre los elementos cambia de una forma no lineal con el número de elementos y la complejidad se incrementa a un ritmo mucho más que lineal.
- A diferencia de los sistemas físicos, no sigue reglas físicas naturales, lo que lo convierte en un sistema abstracto con innumerables relaciones internas difíciles de modelar y controlar.

▼ Qué pasa si intentamos eliminar la complejidad?

Podemos eliminar su esencia. No hay que buscar eliminar la complejidad porque es parte de la esencia del SW

▼ Qué cosas derivan de la complejidad?

No sólo se derivan problemas técnicos, sino también problemas de gestión

Hace difícil tener una visión de conjunto, impidiendo la integridad conceptual.

▼ **Conformidad** → no lo entendí :(

▼ Cuál es la diferencia entre la gente del SW y los físicos o matemáticos que en teoría también trabajan con complejidades

Las labores del físico descansan en un profunda fe de que hay principios unificadores que deben encontrarse.

Nada de esa fe conforta al ingeniero de software. La mayoría de la complejida que el debe controlar es arbitraria, forzada sin ritmo ni razón por las muchas instituciones humanas y sistemas a los que debe ajustarse

▼ De dónde viene la complejidad?

La mayoría de la complejidad viene del tener que ajustarse a otros interfaces; esta complejidad no puede simplificarse solamente con un rediseño del software

▼ **Variabilidad**

▼ A qué está sometido constantemente el SW?

A cambios o presiones para que cambie

▼ Es difícil cambiar SW?

Es fácil que cambie el software → es puro pensamiento, infinitamente maleable

Pero es difícil cambiar el SW → se van introduciendo más errores y complejidad a medida que el sistema crece y evoluciona.

▼ Por qué cambia el SW?

El software vive dentro de un entorno cultural de aplicaciones, usuarios, leyes y hardware. Todo este entorno cambia continuamente, y esos cambios inexorablemente fuerzan a que se produzcan cambios en el software.

- Cuando un SW triunfa → la gente intenta usarlos para nuevas aplicaciones en el límite o más allá del dominio original para el que se diseño
- El software exitoso sobrevive más allá de la vida del hardware para el que fue diseñado → nuevos computadores, nuevos discos, pantallas, impresoras... y el software debe adaptarse para estas nuevas oportunidades

▼ **Invisibilidad**

▼ Por qué decimos que el SW es invisible e in-visualizable?

El software es invisible en el sentido de que no tiene una representación física tangible y espacial.

En el momento que intentamos crear diagramas de las estructuras de software, descubrimos que no está constituido por uno, sino por varios gráficos en distintas direcciones, superpuestos unos sobre otros.

No puede ser visualizado como un artefacto físico (como un edificio o una máquina), lo que hace que sea difícil para los desarrolladores comprender y comunicar su estructura y funcionamiento.

▼ Y en qué influye que el SW sea invisible?

Esta falta de visibilidad dificulta la planificación, el diseño y la depuración del software.

▼ Solución a dificultades accidentales

▼ Qué tipo de dificultad aborda el desarrollo de la tecnología de SW?

Dificultades accidentales, no esenciales → A diferencia de las dificultades esenciales, las dificultades accidentales **pueden reducirse o eliminarse** mediante mejoras en herramientas y metodologías.

▼ Cuáles fueron los avances que abordaron las dificultades accidentales?

▼ **Lenguajes de Alto Nivel**

▼ Qué beneficios hubo en la evolución de lenguajes de bajo nivel a lenguajes de alto nivel?

La evolución de lenguajes de bajo nivel a lenguajes de alto nivel ha simplificado la tarea de escribir código, reduciendo la complejidad asociada a la implementación técnica.

- Productividad, fiabilidad, simplicidad y comprensibilidad
- Libera a un programa de mucha de su dificultad accidental
- Reduce el trabajo intelectual del usuario ya que elimina parte de la abstracción del programa → Los lenguajes más abstractos permiten a los desarrolladores centrarse más en la lógica de negocio y menos en los detalles de hardware

▼ **Tiempo compartido**

▼ Qué beneficios hubo en la aparición del tiempo compartido?

Permite la inmediatez → esto nos permite una visión integral de la complejidad.

Antes, uno se olvidaba de los detalles que tenía en mente en el momento que se dejaba de programa y se preparaba para compilar y ejecutar

▼ Qué problema hay con la evolución del tiempo compartido?

Que la ventaja que aporta es acortar el tiempo → Pero conforme este tiempo de respuesta se aproxima a cero, llega un momento que supera el umbral de percepción humana, que es de unos 100 milisegundos. Más allá de ese límite, no habrá beneficios.

- **Entornos de desarrollo unificados**
  - Qué serían estos entornos de desarrollo?

    Los entornos de desarrollo integrados (IDEs), herramientas de depuración, sistemas de control de versiones y otras herramientas

  - Cuáles fueron los primeros entornos de programación integrados que se difundieron?

    Unix e Interlisp

  - Qué dificultades atacaban?

    Atacaban a las dificultades accidentales que se producen al usar programas individuales de forma conjunta, proporcionando librarías integradas, formatos de fichero unificados, y colas y filtros → FACILITAN LA IMPLEMENTACIÓN PRÁCTICA

Hopes for the silver

- Qué se analiza en este apartado?

  Analiza varias áreas que, en su opinión, ofrecen la posibilidad de avances significativos en el desarrollo de software, aunque no considera que ninguna de ellas sea una "bala de plata" definitiva. Estas áreas son tecnologías o enfoques que podrían mejorar el proceso de desarrollo, aunque **no pueden resolver las dificultades esenciales** del software.

- **Ada y otros avances en lenguajes de alto nivel**
  - Qué fue **ada**?

    Fue un lenguaje de alto nivel de propósito general de los años 80 que revolucionó la época → impulsaba un diseño moderno y modularidad

  - Qué son los MIPS?

Los **MIPS** (Millions of Instructions Per Second) son una medida de rendimiento utilizada en informática para expresar la velocidad a la que una CPU puede ejecutar instrucciones

▼ Ada ha demostrado ser la bala de plata que acabe con el monstruo de la productividad del software?

Si bien los lenguajes de alto nivel, como Ada, han hecho que el desarrollo de software sea más accesible y productivo, no eliminan la complejidad intrínseca del problema que el software intenta resolver.

▼ **Programación orientada a objeto**

▼ Qué permitió el POO?

La POO permite encapsular datos y comportamientos en objetos, lo que mejora la modularidad y facilita la reutilización del código

▼ Qué tipos de datos se distinguen?

- **Tipos de datos abstractos** → el tipo del objeto debería ser definido por un nombre, un conjunto de valores y un conjunto de operaciones propios más que por su estructura almacenada, la cual debería ocultarse
- **Tipos de datos jerárquicos** → nos permiten definir interfaces generales que pueden ser refinadas posteriormente suministrando tipos subordinados

▼ Cuál es la relación entre ambos conceptos?

Los dos conceptos son ortogonales: podemos tener jerarquía sin ocultación u ocultación sin jerarquía.

Ambos conceptos representan avances reales en el arte de construir software.

▼ Qué complejidad eliminan?

Para ambos tipos de abstracción, el resultado es eliminar un tipo de dificultad accidental de un mayor nivel y permitir un mayor nivel de expresión en el diseño.

Pero no pueden eliminar la complejidad esencial del **diseño**

▼ **Inteligencia Artificial**

▼ Espera algo el autor sobre la IA?

No, expresa escepticismo sobre su capacidad para resolver las dificultades esenciales porque las decisiones de diseño son muy dependientes del juicio humano

▼ Qué es Inteligencia artificial?

Dos definiciones:

- *el uso de ordenadores para solucionar problemas que previamente sólo podían solventarse utilizando inteligencia humana*
- *El uso de un conjunto específico de técnicas de programación conocidas como heurísticas o programación basada en reglas. En esta aproximación, se estudia a expertos humanos para determinar que heurísticas o reglas utilizan para solucionar problemas. El programa se diseña para solucionar un problema de la misma forma que los humanos parecen hacerlo*

▼ **Sistemas expertos**

▼ Qué es un sistema experto?

Los sistemas expertos son una forma de IA diseñada para tomar decisiones complejas, utilizando reglas codificadas de un dominio específico

Un sistema experto es un programa que contiene un motor de inferencias generalista y una base de reglas, que coge datos de entrada y suposiciones, explorar las inferencias derivadas de la base de reglas, consigue conclusiones y consejos, y ofrece explicar sus resultados mostrando su razonamiento al usuarios

▼ Para qué son ventajosos?

Estos sistemas ofrecen ventajas claras sobre algoritmos programados diseñados para conseguir las mismas soluciones para los mismos problemas

▼ Qué tiene que ver esto con los motores de inferencia?

Un **motor de inferencia** es un componente fundamental de los **sistemas expertos**. Su función principal es aplicar un conjunto

de reglas a una base de datos de conocimientos para inferir nuevas conclusiones o tomar decisiones

▼ Cuál es el avance más importante que ofrece esta tecnología?

La separación de la complejidad de la aplicación de la la complejidad del programa

▼ Cómo puede aplicarse esta tecnología al trabajo de la ingeniería del software?

- Pueden sugerir las reglas para la interfaz,
- aconsejar sobre estrategias de test,
- recordar las frecuencias de tipos de bugs, y
- mostrar claves para la optimización.

▼ Cuál es la contribución más poderosa de los sistemas expertos?

La contribución más poderosa de los sistemas expertos posiblemente será poner al servicio del programado inexperto la experiencia y visión acumulada por los mejores programadores

▼ **Programación "Automática"**

▼ A qué se refiere con programación “automática”

A la generación de un programa para resolver un problema a partir de las especificaciones del problema

▼ Propiedades favorables de estas aplicaciones

- 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 explicitas para seleccionar técnicas de solución, a partir de parámetros dados por el problema

▼ **Programación gráfica**

▼ A qué se refiere con esto?

A la aplicación de los gráficos de ordenador al diseño de software

▼ Al autor le gusta esto?

No, parece que lo odia con todo su ser. Dice que tiene estos problemas:

- "*el flujograma es una pobre abstracción de la estructura del software*" → "*los programadores dibujan los flujogramas después, no antes de haber escrito los programas que describen*"
- "*las pantallas de hoy son demasiado pequeñas, en pixeles, para mostrar tanto el conjunto como la resolución de cualquier diagrama de software seriamente detallado*"
- "El software es difícil de visualizar" → no se puede representar toda la complejidad

▼ Qué es la analogía VLSI y por qué es errónea?

VLSI se refiere a la tecnología de integración a gran escala que permite incorporar millones de transistores en un solo chip de circuito integrado. Esta tecnología ha permitido avances significativos en el diseño y fabricación de hardware, haciendo posible crear chips más poderosos y complejos.

La analogía sugiere que, al igual que el VLSI ha reducido la complejidad del hardware al integrar más funcionalidades en un solo chip, el desarrollo de software podría beneficiarse de avances similares. Se espera que mejoras en herramientas, metodologías o lenguajes de programación permitan manejar la complejidad del software de manera más eficiente, similar a cómo VLSI ha manejado la complejidad en el hardware.

▼ **Verificación del programa**

▼ En qué consiste?

La verificación del programa busca probar formalmente que el software cumple con sus especificaciones

▼ Es posible encontrar una bala de plata en este ámbito, según el autor?

No, las verificaciones cuestan tanto trabajo que sólo unos pocos programas son verificados

Aunque la verificación podría reducir la carga en el testeo del programa, no puede eliminarla

▼ **Entornos y herramientas**

▼ Que ganancia podemos esperar de las mejoras e investigaciones en los entornos de programación?

Prometen libarnos 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 miriada de detalles que deben ser seguidos de forma precisa por el programador individual y mantenidas uniformemente por el grupo de colaboradores en un único sistema.

▼ **Estaciones de trabajo**

▼ Que ganancia puede esperar el arte del software del incremento rápido y seguro de la potencia y memoria de las estaciones de trabajo individuales?

Seguro que unas estaciones de trabajo más potentes serán bienvenidas. Pero no podemos esperar mágicos avances gracias a ellas.

▼ Ataques prometedores sobre las dificultades esenciales

▼ Se encontró alguna solución mágica al estilo de duplicar la productividad como en el HW?

No, pero hay buenos trabajos y avances en tema

▼ Por qué cosa están limitados los ataques a los accidentes en el proceso de SW?

Por la productividad, que el autor la define como esta ecuación:

$$Time\ of\ task = \sum (Frequency)_i \ x \ (Time)_i$$

▼ Cuáles son las balas de plata prometedoras en la actualidad (ataques sobre la esencia)?

▼ **Comprar vs construir**

▼ Cuál es la alternativa a construir SW?

No construir nada y comprarlo.

Hay un mercado **masivo** de productos de SW → cualquiera de estos productos es más barato de comprar que de hacer

"*un software comprado cuesta lo mismo que un año de programador. Y la entrega es inmediata!*"

▼ Según el chabon este, cuál es el mayor avance en la ingeniería de SW?

El desarrollo del mercado de masas

El uso de **n** copias de un software multiplica de forma efectiva la productividad de sus desarrolladores por **n**. Eso implica una mejora de la productividad enorme.

▼ **Refinamiento de requisitos y prototipado rápido**

▼ Cuál es la parte más dura de hacer SW?

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 la personas, las máquinas y los otros sistemas de software. Ninguna otra parte del trabajo destroza tanto el resultado final si se hace mal. Ninguna otra parte resulta tan difícil de rectificar a posteriori.

▼ Qué sabe el cliente?

Nada, 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

Este chabón asegura que es realmente imposible para un cliente, trabajando con un ingeniero de software, especificar

completa, precisa y correctamente los requisitos exactos de un sistema de software moderno sin probar algunas versiones de prueba del sistema.

▼ Qué promete ser una bala de plata (ataca la complejidad esencial)?

Una de las mayores promesas de los esfuerzos tecnológicos actuales, y una que ataca a la esencia, no los accidentes, del problema del 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.

▼ Qué debe hacer el prototipo?

Simular las interfaces más importantes e implementa las funciones principales del sistema deseado, aunque no se ajuste a las limitaciones de velocidad del hardware, tamaño o coste

▼ Cuál es el propósito de un prototipo?

El propósito del prototipo es hacer real la estructura del concepto del software especificado, para que el cliente pueda comprobar su consistencia y su usabilidad.

▼ **Desarrollo incremental: hacer crecer en vez de construir software**

▼ Qué problema hay con intentar construir todo a la primera (sin incrementos)?

Las escructuras conceptuales que construimos hoy son demasiado complicadas para ser especificadas de forma precisa de forma temprana, y demasiado complejas para ser construidas sin fallos

▼ Analogía con nuestro cerebro

A nuestro cerebro no se le puede hacer un mapa, esto es porque no se **construyó** de una, fue **creciendo** con el tiempo.

▼ Qué implica a nivel práctico el desarrollo incremental?

Esto significa que 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, como acciones o llamadas a subrutinas vacías en el nivel inferior

Cada función añadidad y cada nueva provisión para datos o circunstancias más complejos crece organicamente a partir de lo ya existente.

▼ Qué efectos tiene el desarrollo incremental?

Efectos en la MORAL → El entusiasmo salta cuando hay un sistema en marcha, aunque sea uno simple. Los esfuerzos se redoblan cuando el primer dibujo de un software de gráficos aparece en la pantalla, aunque sea sólo un rectángulo

▼ **Grandes diseñadores**

▼ Cuál es el centro para mejorar el desarrollo de SW?

Las personas 👫

▼ En qué debemos enfocarnos para tener buenos diseñadores?

En promover las BUENAS PRÁCTICAS

▼ Qué diferencia un diseño bueno de uno excelente?

Los diseños excelentes proceden de diseñadores excelentes

▼ Cuál cree el autor que es el esfuerzo individual más potente?

El esfuerzo individual más importante que podemos hacer es desarrollar las formas de que surjan grandes diseñadores

▼ Cómo desarrollar a los diseñadores excelentes?

- Identificar sistemáticamente a los mejores diseñadores tan pronto como sea posible. Los mejores a menudo no son los que tienen más experiencia.
- Asignar un mentor de carrera responsable del desarrollo de su futuro, y realizar un seguimiento cuidadoso de su carrera.
- Idear y mantener un plan de desarrollo de carrera para cada una de las jovenes promesas, incluyendo aprendizaje junto a diseñadores excelentes

cuidadosamente seleccionados, episodios de educación formal avanzada y cursillos, todo entrelazado con asignación de diseños solitarios y liderazgo técnico.

- Dar oportunidades para que los diseñadores en desarrollo interactúen y se estimulen con otros.

▼ Manifiesto Ágil

▼ Aunque valoramos los elementos de la derecha, valoramos más los de la izquierda

1. **Individuos e interacciones** → sobre procesos y herramientas
2. **Software funcionando** → sobre documentación extensiva
3. **Colaboración con el cliente** → sobre negociación contractual
4. **Respuesta ante el cambio** → sobre seguir un plan

▼ Doce principios

1. Nuestra mayor prioridad es **satisfacer al cliente** mediante la entrega temprana y continua de software con valor.
2. Aceptamos que los **requisitos cambien**, incluso en etapas tardías del desarrollo. Los procesos Ágiles aprovechan el cambio para proporcionar ventaja competitiva al cliente.
3. **Entregamos** software funcional **frecuentemente**, entre dos semanas y dos meses, con preferencia al periodo de tiempo más corto posible.

4. Los responsables de negocio y los desarrolladores **trabajamos juntos** de forma cotidiana durante todo el proyecto.
5. Los proyectos se desarrollan en torno a **individuos motivados**. Hay que darles el entorno y el apoyo que necesitan, y confiarles la ejecución del trabajo.
6. El método más eficiente y efectivo de comunicar información al equipo de desarrollo y entre sus miembros es la **conversación cara a cara**.
7. El **software funcionando** es la medida principal de progreso (y el **éxito**).
8. Los procesos Ágiles promueven el **desarrollo sostenible**. Los promotores, desarrolladores y usuarios debemos ser capaces de mantener un ritmo constante de forma indefinida.
9. La atención continua a la **excelencia técnica** y al **buen diseño** mejora la Agilidad.
10. La **simplicidad**, o el arte de maximizar la cantidad de trabajo no realizado, es **esencial**.
11. Las mejores arquitecturas, requisitos y diseños emergen de **equipos auto-organizados**.
12. A intervalos regulares el equipo reflexiona sobre cómo ser más efectivo para a continuación **ajustar** y perfeccionar **su comportamiento** en consecuencia.