

# Frederick P. Brooks No Silver Bullet Essence and Accident in Software Engineering

Grupo 3

Ingeniería de Software II - 1er Cuatrimestre 2016

June 16, 2016

# Qué vamos a ver

- 1 ¿Silver Bullet?
- 2 Dificultades esenciales vs. Dificultades accidentales
- 3 Esencia
  - Propiedades inherentes a la esencia del software
- 4 Avances pasados sobre dificultades accidentales
- 5 Hopes for the Silver
- 6 Promising Attacks
- 7 ¿Qué pasó después del paper?

# ¿Silver Bullet?

- Un proyecto de software es como un hombre-lobo: usualmente inocentes y familiares, pero con la capacidad de convertirse en un monstruo.
- Buscamos una bala de plata que disminuya drásticamente los costos de producir software.
- Sin embargo, por la naturaleza misma del software, no hay ningún avance en la tecnología ni en las técnicas de gestión que por sí mismo prometa un aumento en productividad, confiabilidad y simplicidad de siquiera un orden de magnitud, de la misma manera que ocurre con el hardware.

# Dificultades esenciales vs. Dificultades accidentales

Dos tipos de dificultades, vistas a través de un prisma aristotélico:

- **Esenciales:** inherentes a la naturaleza del software; la lógica para resolver el problema.
- **Accidentales:** que tienen que ver con la producción del software y que no son inherentes; el lenguaje de programación, o el entorno de desarrollo.

- **Construcción de conceptos entrelazados:** conjuntos de datos, relaciones entre elementos de datos, algoritmos e invocaciones de funciones.
- **Abstracta:** la construcción conceptual es la misma bajo muchas diferentes representaciones.

**La parte difícil de contruir software es la especificación, diseño y testeo de la construcción conceptual, no representar y probar la fidelidad de la representación.**

- **Complejidad:**

- Las entidades de software son más complejas en tamaño que cualquier otra construcción humana.
- Los sistemas tienen gran número de estados lo que los hace difícil de concebir, describir y probar.
- Escalar software no es repetir elementos con tamaño más grande: es necesario incrementar la cantidad de elementos (ergo, sus interacciones, y su complejidad).

- **Conformidad:**

- Mucha de la complejidad es arbitraria y viene de la mano de las interfaces con las que un sistema debe conformar.

## ● **Modificabilidad:**

- El software de un sistema encarna su función, que es la parte que más siente el peso del cambio, en parte porque es extremadamente maleable.
- A veces el cambio viene porque surgen nuevos casos que no pertenecían al dominio original.
- Otras, porque el software sobrevive el ciclo de vida de la máquina para la cual fue escrito.

## ● **Invisibilidad:**

- El software es invisible e invisualizable.
- No tiene una representación espacial.
- Diagramar una estructura de software involucra diversos diagramas que miran cosas distintas.
- Priva a la mente de usar algunas de sus herramientas conceptuales más potentes y dificulta el diseño y la comunicación.

- **Lenguajes de Alto Nivel:**

- Eliminó todo un nivel de complejidad que no era inherente al programa al encarnar las contrucciones que se querían en el programa abstracto y al ignorar los detalles de más bajo nivel.

- **Time-Sharing:**

- Preserva la inmediatez, y nos permite tener en la mira a la complejidad.
- Disminuye el tiempo de respuesta del sistema.
- Si sigue evolucionando, eventualmente va a ser imposible notarlo y no se obtendrán más ventajas

- **Ambientes de desarrollo unificados:**

- Atacan la complejidad accidental de usar programas en conjunto al proveer bibliotecas integradas, formatos unificados, etc.



- **Ada y otros avances en lenguajes de alto nivel:**

- Ya hablamos de los lenguajes de alto nivel.
- El gran avance de Ada tiene que ver con su filosofía de modularización, tipos de datos abstractos y estructuración jerárquica.

- **Programación Orientada a Objetos:**

- Dos conceptos diferentes: tipos de datos abstractos y tipos jerárquicos (o clases).
- Remueven una dificultad accidental al permitirle al diseñador expresar la esencia de su diseño sin tener que usar mucha sintaxis que no suma información.
- Eliminan las dificultades accidentales de la expresión del diseño a un mayor nivel, pero no la complejidad del diseño.

- **Inteligencia Artificial:** (dos definiciones, según Parnas)
  - AI-1: usar las computadoras para resolver problemas que sólo se podían resolver con inteligencia humana.
  - AI-2: determinar reglas y heurísticas usadas por expertos para resolver problemas, y diseñar un programa para que los resuelva de la misma manera.
  - Problema AI-1: algo es AI-1 hoy, pero cuando veamos cómo funciona el programa, y entendamos la solución, deja de ser AI. Además, las soluciones suelen ser específicas de un problema (image recognition, speech recognition), y no universales.

- **Sistemas Expertos:** (AI-2)

- Un programa con un motor de inferencia y una regla base. Se le dan datos de entrada y presunciones.
- Infiere (probabilística o determinísticamente), y devuelve conclusiones y consejos, con el camino para llegar a ellos.
- Su poder está en bases de datos que reflejen cada vez mejor el mundo real.
- ¿Aplicación al software? Poder sugerir reglas de interfaz, aconsejar estrategias de testing, recordar frecuencias de tipos de bugs, ofrecer pistas de optimización.

- **Programación Automática:**

- Generar un programa que solucione un problema, indicando la especificación del problema.
- Según Parnas, eufemismo para “lenguaje de nivel más alto que los disponibles hasta ahora”.
- Hay algunas aplicaciones (sorting, ecuaciones diferenciales), pero no generalizables al mundo del desarrollo de software.

## ● Programación gráfica:

- Uso de estructuras gráficas para desarrollar software.
- Flowchart: abstracción pobre de una estructura de software.
- (Queja obsoleta) Pantallas chicas y con pocos pixeles para mostrar un diagrama detallado.
- Es muy difícil visualizar software, y una dimensión no alcanza para una visión global.

## ● Verificación de Programas:

- Eliminar errores en la fase de diseño.
- Concepto poderoso, pero lleva mucho trabajo aplicarlo.
- No implica que los programas no tengan errores, sino que el software cumple su especificación. ¿Y si la especificación está mal?
- La parte más difícil es arivar a una especificación completa y consistente.

- **Entornos y Herramientas:**

- Es una ayuda para evitar errores sintácticos y semánticos.
- Pueden ofrecer al programador constantemente detalles a tener en cuenta del sistema, evitándole tener que recordarlos en su memoria.
- Sin dudas mejora la productividad, pero la mejora es marginal

- **Workstations:**

- El cuello de botella va a seguir estando en la cabeza del programador

# Promising Attacks

- **Buy, don't build:** Producir software es costoso, a veces es más conveniente comprar software ya desarrollado.
- Refinamiento de requerimientos y prototipado rápido.
- Cambio de metáfora: De contruir software a **crecer software**. Tomar la autopista del desarrollo iterativo e incremental.
- Mantener buenos diseñadores: **Se puede enseñar a diseñar bien, pero no a ser un excelente diseñador.** Hay que identificar a los grandes diseñadores, son tan importantes como los gerentes.

# ¿Qué pasó después del paper?

## “No Silver Bullet” Refired

- Mejoras en productividad, pero no en órdenes de magnitud.
- Buy, don't build: esto parece cumplirse. Uno no implementa todos los aspectos de un sistema.
- Desarrollo incremental, iterativo con feedback de usuarios: hoy se usa mucho. Diversas metodologías en la actualidad adoptan esta filosofía.
- Reusar código (buen diseño, buena documentación, vocabulario común): es mejora marginal, pero ayuda.
- OOP: ayuda, pero no cambió al mundo... todavía
- Mantener buenos diseñadores: son rockstars. Identificarlos, retenerlos, mimarlos. Todavía falta énfasis en este aspecto.

### F.P. Brooks

La complejidad es el negocio en el que estamos, y la complejidad es lo que nos limita.