No Silver Bullet

Abstract

Todo desarrollo de software se basa en el embellecimiento de las estructuras conceptuales que componen la entidad abstracta del software (aca recae las "accidental tasks") y la representacion de las mismas en un lenguaje que pasa a ser codigo de maquina con sus respectivas limitaciones de velocidad y espacio

La mayoria de las ganancias relacionadas a la productividad consiste en eliminar las barreras artificiales que generan accidental tasks (tareas accidentales)

Algunos ejemplos:
- Limitaciones de Hardware
- Lenguajes de programación raros
- Falta de tiempo de maquina

¿Cuanto de lo que hace un Ingeniero de software recae en lo accidental envez de lo esencial?

Recomendaciones para quien quiera embellecer un gran sistema complejo

- No construir lo que puede ser comprado
- Usar prototipado veloz como parte de iteracion planeada
- Crecer sw. organicamente, añadiendo funciones al usar el software al ser "run, used and tested"
- Identificando y desarrollando los grandes diseñadores conceptuales de la nueva generación

Introducción

Hombres Lobo → dan terror —(¿que nos hace buscar?)→ una bala de plata que lo elimine

El software es similar, aparenta inocencia y puede convertirse en un monstruo. Inocentemente buscamos una bala de plata para bajar los costos tanto como baja el costo de Hardware

Aun así hay un camino posible, modificar la teoría del demonio por la del germen; Eliminando así las aspiraciones a "soluciones mágicas"

¿Tiene que ser tan difícil? — Dificultades Esenciales

No podemos pretender encontrar el equiv en SW de lo que lograron la electrónica, transistores e Integración a gran escala en el HW.

↳ El avance del HW es y fue una anomalía

Para entender el "rate of progress" de una tecnología de SW debemos examinar sus dificultades.

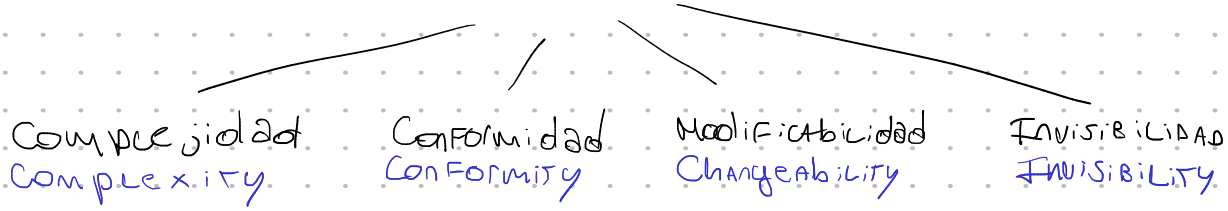
Dos tipos de dificultades (parafraseando a Aristoteles)
- Esencia ↦ Dif. Inherentes a la naturaleza del software
- Accidentes ↦ Dif. Que recaen en la producción

La parte mas difícil es la especificación, diseño y testeo del constructo conceptual, no el representarlo y testear la fidelidad de la representación

Si esto es cierto construir software siempre va a ser difícil

No va a haber bala de plata

Tengamos en cuenta las propiedades inherentes de la esencia irreducible del SOFTWARE



Complejidad

gran numero de estados → Complejiza concebirlo / describirlo y testearlo

→ Tienen muchisimos mas ordenes de magnitud de estados que una computadora

El SOFTWARE escala complejidad de forma NO lineal

A diferencia de la fisica ó matematica No se pueden ignorar las pequeñeses y simplificar el modelo ya que las complejidades aca son la esencia

Los problemas que da la complejidad son → Tecnicos de Managing

Conformidad

El SOFTWARE a veces que ajustarse porque recien llega a la escena. En otros pq a veces es lo mas correcto.

Pero en su mayoria de casos la complejidad reciele en tener que adaptarse a otras interfaces. Esto hace que no se pueda solucionar siquiera con un rediseño del SOFTWARE solo.

Cambiabilidad

La entidad de software esta supeditada a la constante presión del cambio. Gran diferencia con lo manufacturado que una vez lanzado, rara vez supone un cambio

Todo software exitoso atraviesa cambios

Invisibilidad

El software es invisible e invisualizable. Si haces un edificio chueco, lo ves a simple vista. En el software eso no pasa. La realidad del software no esta atada al espacio. Incluso al graficar cosas como dependencias, estados, data, etc; su proceso/ejecución no es plano ni jerarquico.

Establecer control conceptual nos permite al menos lograr gráficos jerarquicos, pero esto se logra simplificando o recortando su estructura.

Esos recortes quedan invisibilizados

Past Breakthroughs Solved Accidental Difficulties

Avances previos resolvieron complejidades accidentales, no las esenciales

Lenguajes de alto nivel
- fue x5 productividad. Gran avance
- fue + Reliability + Simplicidad + Comprensibilidad
- fue Libera al programa de complejidad accidental
- fue Hay un punto donde hay tanto nuevo datatype y constructores ocultos que el alto nivel termina empeorando el labor intelectual para quien no usa sus constructores esotericos

TIME SHARING → Mejora productividad (no tanto como los leng de alto nivel) y calidad del producto

→ Preserva inmediatez, nos da una overview de la complejidad

Entornos de desarrollo unificado → mejora de entornos de progr. son un avance importante y son fuertemente investigados

Esperanzas de la plateada

Ada → Filosofía de Modularización
→ Tipos abstractos de datos
→ Estructura jerárquica

No es una bala de plata pero cree que es buen camino

Object Oriented Programming
- La dos ideas
  - Tipos abstractos de datos (modula/ada) → obj def por nombre y set de valores y operaciones
  - Tipos jerárquicos
    - Simula 67
    - Def de interfaces que pueden ser redefinidas por tipos subordinados

Estos avances resuelven lo accidental, no lo esencial

Inteligencia Artificial → No cree del todo en que genere suficiente ganancia

↓

Expert Systems

An expert system is a program containing a generalized inference engine and a rule base, designed to take input data and assumptions and explore the logical consequences through the inferences derivable from the rule base, yielding conclusions and advice, and offering to explain its results by retracing its reasoning for the user. The inference engines typically can deal with fuzzy or probabilistic data and rules in addition to purely deterministic logic.

→ Separa complejidad de la aplicación de la complejidad del programa en sí

→ Básicamente predice copiar

AUTOMATIC PROGRAMMING →

In short, automatic programming always has been a euphemism for programming with a higher-level language than was presently available to the programmer.[8]

– Parnas

Exceptions can be found. The technique of building generators is very powerful, and it is routinely used to good advantage in programs for sorting. Some systems for integrating differential equations have also permitted direct specification of the problem. The system assessed the parameters, chose from a library of methods of solution, and generated the programs.

- These applications have very favorable properties:
- The problems are readily characterized by relatively few parameters.
- There are many known methods of solution to provide a library of alternatives.
- Extensive analysis has led to explicit rules for selecting solution techniques, given problem parameters.

It is hard to see how such techniques generalize to the wider world of the ordinary software system, where cases with such neat properties are the exception. It is hard even to imagine how this breakthrough in generalization could conceivably occur.

GRAPHICAL PROGRAMMING → COMPARA CON DIAGRAMAS DE HARDWARE

→ NO LE DA MUCHA BOLA

PROGRAM VERIFICATION → PUEDE SER LA BALA PLATEADA EL ELIMINAR LOS ERRORES EN LA FUENTE, EN LA FASE DE DISEÑO?

NO

NO ES MAGICA Y REQUIERE LABOR

→ NO SIGNIFICA QUE DE "ERROR PROOF PROGRAMS"

AMBIENTE Y HERRAMIENTAS → IMPORTANTE PERO SU OPTIMIZACION ES MARGINAL

WORKSTATIONS → MEJORES MAQUINAS ← RAPIDA COMPILACIÓN
RAPIDA EDICIÓN DE DOCUMENTOS

↓

ESTA BUENO, PERO NO ES MAGIA NI PODEMOS PRETENDER QUE LO SEA

Promising Attacks on the Conceptual Essence

All of the technological attacks on the accidents of the software process are fundamentally limited by the productivity equation:

$$Time\ of\ task = \Sigma\,(Frequency)_i \times (Time)_i$$

Buy versus Build → Solución a construir software, no construirlo

→ Many users now operate their own computers day in and day out on varied applications without ever writing a program. Indeed, many of these users cannot write new programs for their machines, but they are nevertheless adept at solving new problems with them.

I believe the single most powerful software productivity strategy for man organizations to day is to equip the computer-naïve intellectual workers on the firing line with personal computers and good generalized writing, drawing, file and spreadsheet programs, and turn them loose. The same strategy, with simple programming capabilities, will also work for hundreds of laboratory scientists.

En parte es lo que pasa en muchas áreas

Requerimientos refinados y prototipado veloz

→ Lo mas difícil de hacer SW, es decidir que construir precisamente

→ No hay parte mas difícil que esta para arreglar despues

→ I would go a step further and assert that it is really impossible for clients, even those working with software engineers, to specify completely, precisely, and correctly the exact requirements of a modern software product before having built and tried some versions of the product they are specifying.

↳ Herramientas de prototipado rapido acortan este camino

Much of present-day software acquisition procedures rests upon the assumption that one can specify a satisfactory system in advance, get bids for its construction, have it built, and install it. I think this assumption is fundamentally wrong, and that many software acquisition problems spring from that fallacy. Hence they cannot be fixed without fundamental revision, one that provides for iterative development and specification of prototypes and products.

Incremental development - grow, not build, software

↳ writing => building => growing software

↳ Software should be grow by incremental development

↳ En principio debe correr, llamar a los subprogramas y de ahi incrementarlo de a poquito haciendo de los subprogramas acciones.

↳ TOP-DOWN design

Great Designers

- La cuestion de como mejorar las "Software Art centers" siempre recae en las personas
- Buenas practicas de diseño se pueden enseñar
- Lo mas importante es formar Buenos diseñadores

My first proposal is that each software organization must determine and proclaim that great designers are as important to its success as great managers are, and that they can be expected to be similarly nurtured and rewarded. Not only salary, but the perquisites of recognition–office size, furnishings, personal technical equipment, travel funds, staff support–must be fully equivalent.

How to grow great designers? Space does not permit a lengthy discussion, but some steps are obvious:

- Systematically identify top designers as early as possible. The best are often not the most experienced.
- Assign a career mentor to be responsible for the development of the prospect, and keep a careful career file.
- Devise and maintain a career development plan for each prospect, including carefully selected apprenticeships with top designers, episodes of advanced formal education, and short courses, all interspersed with solo design and technical leadership assignments.
- Provide opportunities for growing designers to interact with and stimulate each other. ■