UNIVERSIDAD TECNOLÓGICA NACIONAL
FACULTAD REGIONAL CÓRDOBA
INGENIERÍA EN SISTEMAS DE INFORMACIÓN
CÁTEDRA DE INGENIERÍA DE SOFTWARE

INGENIERÍA DE SOFTWARE

Trabajo Conceptual N°1: Informe Técnico

CAUSAS, SÍNTOMAS Y RELEVANCIA ACTUAL DE LA CRISIS DEL SOFTWARE

Versión **1.1**

Miércoles 1° de mayo de 2019

AÑO **2019**
COMISIÓN **4K4**

DOCENTES:

COVARO, LAURA INÉS (ADJUNTO)

ROBLES, JOAQUÍN LEONEL (JTP)

BELLI, GIULIANA (AYUDANTE 1RA)

AUTORES:

ARÉVALO, OSCAR RUBÉN (52340)

CAMPANA, ESTEBAN ROQUE (54843)

CUADRADO, ALEXIS (61022)

GARCÍA PIÑAS, GRICEL (50885)

IRAHOLA, MAURICIO ALEJANDRO (38618)

JORNET, PABLO FEDERICO (62901)

Causas, síntomas y relevancia actual de la Crisis del Software

Arévalo, Oscar Rubén; Campana, Esteban Roque; Cuadrado, Alexis; García Piñas, Gricel; Irahola, Mauricio Alejandro; Jornet, Pablo Federico

Universidad Tecnológica Nacional, Facultad Regional Córdoba

Ingeniería en Sistemas de Información - Cátedra de Ingeniería de Software

Abstract – **Hacia fines de los años 60' se hizo evidente la existencia de una "Crisis del Software", que llevó a los expertos de la época a adoptar un nuevo enfoque para el desarrollo, dando origen a la Ingeniería de Software como disciplina profesional. Analizamos en este documento los síntomas de la Crisis, sus causas y explicamos por qué son relevantes aún en la actualidad.**

Palabras Clave

Crisis del Software, Ingeniería de Software, Complejidad, Calidad, Sobrecosto, Entrega tardía, Defectos, Mantenibilidad, Requerimientos insatisfechos, Vaporware, Spaghetti Code, GOTO, Bauer, Dijkstra, Brooks.

Introducción

En 1968 y 1969, el Comité de Ciencias de la Organización del Tratado del Atlántico Norte (OTAN) patrocinó las primeras conferencias internacionales de Ingeniería de Software, llevadas a cabo en Garmisch, Alemania, con el propósito de discutir lo que F.L. Bauer y E.W. Dijkstra denominaron "Crisis del Software" [1]. Este término fue acuñado para describir la situación que atravesaba la industria a fines de los años 60', década durante la cual se acentuaron las dificultades asociadas al desarrollo de sistemas de software complejos. Entregas tardías, costos demasiado altos, requerimientos insatisfechos, baja calidad y documentación escasa son algunos de los síntomas de los proyectos de software que evidenciaron la existencia de esta crisis y motivaron a los profesionales más sobresalientes de la época a reunirse para analizar las raíces de estos problemas y considerar las posibles soluciones.

Síntomas

Para 1968, la Crisis del Software se había manifestado de diversas maneras:

1. Muchos proyectos excedían el presupuesto establecido.
2. Muchos proyectos presentaban demoras de meses e incluso años.
3. Frecuentemente, el software no era siquiera terminado o entregado.
4. En general, los productos de software presentaban muchos defectos y eran ineficientes.
5. La funcionalidad entregada rara vez satisfacía los requerimientos de los usuarios finales.
6. Los proyectos grandes eran extremadamente difíciles de gestionar.
7. Una vez creado, el software era casi imposible de mantener.

A continuación, abordamos algunos de los problemas más característicos de la Crisis del Software derivados de los síntomas anteriores y analizamos brevemente sus causas.

I. Sobrecostos y demoras

Un problema clásico de los proyectos de software, evidenciado durante la Crisis del Software, es el de la finalización tardía de los proyectos, frecuentemente sobrepasando los límites de presupuesto. Uno de los casos más emblemáticos de la historia fue el del sistema operativo IBM System/360. Fue un proyecto colosal que involucró a más de 1000 personas y costó alrededor de 5 mil millones de dólares, lo que representaba el doble de los ingresos anuales de IBM en ese

entonces [5]. En su concepción, el presupuesto para el desarrollo de System/360 era de 25 millones de dólares, suficiente para cubrir los gastos de un equipo de 12 diseñadores y 60 programadores. Se calcula que, en el lapso de 3 años, se empleó un esfuerzo de más de 5000 años-persona en el diseño, implementación y documentación del sistema operativo, que se entregó finalmente un año más tarde de lo previsto. Pese a todo, System/360 se convirtió en un éxito de ventas y permitió a IBM dominar el mercado de mainframes durante las décadas siguientes. Frederick Brooks, el líder de desarrollo del proyecto, inmortalizó muchas de las lecciones aprendidas en su famoso libro The Mythical Man-Month, siendo la más importante de ellas quizás la conclusión de que "agregar gente a un proyecto atrasado, lo atrasa aún más".

II. Software no entregado

Comúnmente, los proyectos de software eran planificados de acuerdo a expectativas y estimaciones poco realistas, lo cual repercutía en demoras de años o meses, en el mejor de los casos. A menudo, esto derivaba en la cancelación definitiva de los proyectos sin haber entregado siquiera una porción de la funcionalidad prometida, ocasionando grandes pérdidas para las empresas que los llevaban adelante y perjudicando a sus clientes. Más aún, no era extraño que las compañías de software anuncien públicamente sus productos meses e incluso años antes de la fecha propuesta de entrega, sin conocer exactamente cómo los desarrollarían o incluso saber si contarían con los recursos necesarios. En relación a esto, se hizo frecuente en la jerga de la industria el uso término vaporware [6], que designa en forma despectiva a un producto de software que ha sido anunciado oficialmente, pero cuyas posibilidades de ser efectivamente desarrollado y entregado son inciertas. Muchas veces el vaporware era publicitado en forma deliberada con la intención de retener a los clientes y prevenir que recurran a la competencia.

Uno de los ejemplos más notorios de vaporware en la historia, ya en la década de los 80, es el de Ovation, una suite de ofimática diseñada para competir con el hasta entonces líder del mercado Lotus 1-2-3. La empresa desarrolladora, Ovation Technologies, recaudó millones de dólares en capital gracias a espectaculares demostraciones que alimentaron la expectativa del público. Sin embargo, fueron incapaces de llevar adelante el desarrollo del producto y tuvieron que admitir que nunca existió [7], lo que culminó con la compañía declarándose en bancarrota en 1984.

III. Complejidad

Hacia fines de los años 60', los costos de fabricación del hardware se habían abaratado drásticamente y continuaban cayendo de forma exponencial. Por el contrario, el costo de desarrollo del software aumentaba a un ritmo similar.

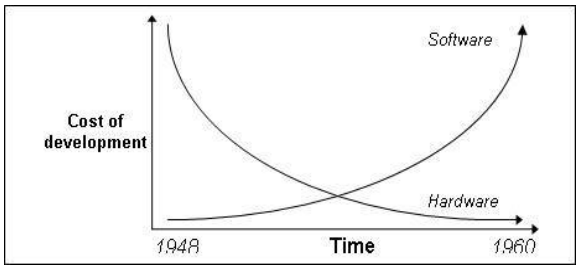

Figura 1: Costo de desarrollo de software vs hardware entre 1948 y 1960 [3]

El hardware no solo era cada vez más barato, sino además potente. La emergencia de grandes mainframes de propósito general con extraordinarias capacidades de cómputo, tales como el IBM 360, hizo posible concebir el desarrollo de sistemas de software cada vez más complejos. Este ritmo vertiginoso con el que crecía el poder de cómputo y con él la complejidad de los problemas que podían ser resueltos con el software, ocasionó que las técnicas y métodos hasta entonces empleados en el

desarrollo de software se vuelvan insuficientes. Lo resume elocuentemente Dijkstra en su artículo The Humble Programmer (El programador humilde): “La mayor causa de la crisis del software es que las máquinas se han vuelto varios órdenes de magnitud más poderosas. Para decirlo sin rodeos: mientras no había máquinas, la programación no era un problema; cuando pasamos a tener solo unas pocas débiles computadoras, la programación se convirtió en un problema menor, y ahora que tenemos computadoras gigantes, la programación se ha vuelto un problema igual de gigante.” [2]

IV. Mantenibilidad

Debido a la ausencia de un enfoque estructurado para el desarrollo de software en general, a la falta de reglas o guías de estilo de programación, y en muchos casos a la falta de pericia técnica y experiencia para lidiar con la complejidad creciente, el software producido era muy difícil de mantener. El código fuente de los programas era a menudo enrevesado y difícil de comprender incluso hasta para sus propios autores. Hacia principios de los 80’ era habitual el uso de la frase peyorativa spaghetti code para describir código de estas características, en donde el flujo de control del programa era tan enredado que podía trazarse un paralelismo con la comida de origen italiano.

Si bien es que la producción de spaghetti code puede deberse a varios factores (principalmente a malas decisiones de diseño), muchos profesionales de la industria se pusieron de acuerdo en que un factor habitual el sobreuso de la sentencia goto, encontrada en muchos de los lenguajes procedurales comúnmente utilizados en la época, tales como FORTRAN. Probablemente la crítica más resonante de dicha sentencia sea la que elaboró Dijkstra en su famosa carta titulada GOTO Considered Harmful (GOTO considerado dañino).

Independientemente de las causas del spaghetti code, lo cierto es que introducir cambios en programas de este tipo resultaba generalmente muy costoso, sino imposible. Es por esto que habitualmente las empresas incurrían en mayores gastos durante el mantenimiento del software que durante su desarrollo.

V. Calidad

A medida que los programas de computación comenzaron a adquirir más protagonismo en el mundo de los negocios, ocasionando que el desarrollo de software se convierta en un verdadero emprendimiento comercial, los clientes se volvieron cada vez más exigentes. En pos de lograr una ventaja competitiva, empezaron a demandar software con mayor frecuencia. Era común en este marco que los desarrolladores, abrumados por las restricciones impuestas y por la falta de metodologías y herramientas de trabajo que se ajusten a las nuevas exigencias, resignen calidad y produzcan software repleto de defectos para cumplir con los plazos y presupuesto establecidos. Por supuesto, las expectativas de los clientes acerca del rendimiento, robustez y tolerancia a fallos de los sistemas crecieron a la par de la demanda, llevando a que muchos proyectos, aun habiendo consumido mucho esfuerzo y dinero, terminen siendo cancelados o considerados un “fracaso”.

VI. Vida y muerte

En relación a lo anterior, a medida que el software se fue haciendo más invasivo, introduciéndose en la mayoría de los aspectos de la vida cotidiana, los riesgos asociados a la baja calidad del software se multiplicaron. En muchos dominios, la presencia de defectos en el software puede ser la diferencia entre la vida y la muerte. Tal es el rubro de la medicina. Entre 1985 y 1987, la máquina de radioterapia Therac-25, producida por una empresa estatal canadiense llamada Atomic Energy of Canada Limited (AECL), ocasionó al menos

seis accidentes en donde les aplicó sobredosis de radiación a distintos pacientes como consecuencia de un fallo en su software de control. Tres de ellos fallecieron. Tras realizarse una investigación se determinó que ACL nunca había probado la combinación completa de software y hardware que constituía el Therac-25 hasta que fue ensamblado para su uso en el hospital.

Ingeniería de Software

Toda crisis representa una oportunidad para el cambio, y los gurúes del software reunidos entonces en Garmisch tenían más que claro que, para hacer frente a estos problemas, el desarrollo de software debía empezar a ser abordado con un enfoque ingenieril, esto es, de una manera sistemática y estructurada, empleando teorías, métodos y herramientas diseñados especialmente para producir y entregar sistemas de software de calidad en tiempo y forma.

El resultado de las conferencias de la OTAN fueron dos extensivos reportes que sentaron las bases de cómo el software debía ser desarrollado aplicando prácticas de la ingeniería, marcando el hito fundante de la Ingeniería de Software como disciplina profesional. En palabras del propio Dijkstra: "La programación empezó como una artesanía practicada de manera intuitiva. Para 1968, existía un acuerdo general en que los métodos de desarrollo de programas utilizados hasta el momento eran inadecuados […] Cuando los métodos entonces utilizados fueron reconocidos como fundamentalmente inadecuados, un nuevo estilo de diseño fue desarrollado; un estilo en donde el programa y su corrección eran diseñados a la par. Esto representó un paso dramático hacia adelante." [3]

Actualidad

Desde el nacimiento de la Ingeniería de Software como disciplina profesional hasta hoy, una gran cantidad de técnicas, procesos y metodologías fueron concebidos con el propósito de aumentar la productividad, simplicidad y eficacia en los proyectos de software. No obstante, como sostiene F. Brooks, no existen las "balas de plata" [8]. El paradigma orientado a objetos nos ayuda a analizar, modelar y programar software utilizando conceptos y abstracciones más familiares, las plataformas de computación en la nube nos permite desentendernos de la configuración y mantenimiento del hardware y el software de base y enfocarnos en el desarrollo del software de aplicación, y las metodologías ágiles nos enseñan a involucrar al cliente para entender mejor sus necesidades y gestionar mejor los cambios. Estas innovaciones, al igual que tantas otras, han logrado reducir la complejidad accidental del software, pero no su complejidad esencial. En consecuencia, muchos (sino todos) los síntomas evidenciados durante la Crisis del Software siguen vigentes aún el día de hoy: los proyectos son difíciles de gestionar, se atrasan, cuestan más de lo previsto, producen software que no satisface las necesidades reales de los usuarios o producen software defectuoso cuyas fallas, en ocasiones, cuestan vidas humanas.

Conclusión

Los problemas tan familiares que llevaron a la Crisis del Software reconocida a fines de la década del 60', tienen raíz en la insuficiencia de los métodos e incapacidad de los profesionales de la industria de entonces para manejar el crecimiento frenético de la complejidad de los problemas que podían ser resueltos por el software; crecimiento que fue facilitado por el abaratamiento del hardware e incremento exponencial de su poder de cómputo. El reconocimiento de la existencia de estos problemas y limitaciones durante las conferencias de la OTAN en el 68' y el 69' permitió llegar a la conclusión de que era indispensable un cambio de enfoque: para hacer frente a la creciente complejidad y

producir software de calidad, en el tiempo y con los límites de presupuesto establecidos, era necesario aplicar principios y metodologías de la ingeniería al desarrollo de software. Esto constituyó el hito fundante de la Ingeniería de Software como disciplina profesional, y permitió la proliferación de herramientas, técnicas, métodos y procesos durante las décadas siguientes, que permitieron aumentar la productividad y la eficacia en el desarrollo de software, aún sin proporcionar "balas de plata".

Por último. si bien es cierto que cuando se menciona la Crisis del Software se hace referencia a un contexto histórico determinado, no se debe presumir los problemas de entonces no son relevantes en la actualidad. Por el contrario; la industria del software vive en constante estado de crisis. La complejidad de los sistemas de software continúa creciendo más rápido que los métodos para manejarla, y los problemas de antaño se presentan en innumerables proyectos el día de hoy. Para evitarlos, es menester que los profesionales contemporáneos reconozcan este hecho y apliquen las enseñanzas que 50 años de Ingeniería de Software nos han dejado.

Referencias

[1] Randell, Brian; Naur, Peter; Buxton J.N. "Software engineering: Concepts and techniques: proceedings of the NATO conferences". Primera edición. Petrocelli/Charte. 1976.

[2] Dijkstra, Edger. "EDW 340: The Humble Programmer". Commun ACM 15. 1972.

[3] "Information Systems and Strategy, Session 2, The Software Crisis". Euromed Marseille School of Management. Recuperado de http://www.chris-kimble.com/Courses/World_Med_MBA/Software_Crisis.html.

[4] Sommerville, Ian. "Software Engineering". Novena Edición. Addison-Wesley. 2011.

[5] Liu, Beyang. "What we can learn from the IBM System/360, the first modular, general-purpose computer". 2016. Recuperado de https://about.sourcegraph.com/blog/the-ibm-system-360-the-first-modular-general-purpose-computer

[6] Dyson, Esther. "Vaporware". 1983. Recuperado de https://cdn.oreillystatic.com/radar/r1/11-83.pdf

[7] Flynn, Laurie. "The Executive Computer". The New York Times. 1995. Recuperado de https://www.nytimes.com/1995/04/24/business/information-technology-the-executive-computer.html

[8] Brooks, Fred. P. "No Silver Bullet – Essence and Accident in Software Engineering". 1987. Recuperado de http://faculty.salisbury.edu/~xswang/Research/Papers/SERelated/no-silver-bullet.pdf

[9] Mack, Chris A. "How to write a good scientific paper". Primera edición. Society of Photo-Optical Instrumentation Engineers. 2018.