ANNIVERSARY EDITION WITH FOUR NEW CHAPTERS

ESSAYS ON SOFTWARE ENGINEERING

THE MYTHICAL MAN-MONTH

FREDERICK P. BROOKS, JR.

Photo credit: ©Jerry Markatos

ACERCA DEL AUTOR

Frederick P. Brooks, Jr., es Profesor Kenan de Ciencias Computacionales de la Universidad de Carolina del Norte en Chapel Hill. Es mejor conocido como “el padre de la IBM Sytem/360,” se desempeñó como su director de desarrollo y más tarde como director de diseño de software del Operative System/360. Por este trabajo, en 1985, fue premiado junto a Bob Evans, y Erich Bloch con la Medalla Nacional de Tecnología. Previamente trabajó como arquitecto de las computadoras IBM Stretch y Harvest.

El Dr. Brooks fundó el Departamento de Ciencias Computacionales en Chapel Hill y lo presidió de 1964 a 1984. Ha servido en el Consejo Nacional de Ciencia y en el Consejo de Ciencia de la Defensa. Actualmente su actividad docente y de investigación se centra en la arquitectura de computadoras, las gráficas moleculares, y los ambientes virtuales.

El Mítico Hombre-Mes

Ensayos acerca de la Ingeniería de Software

Edición de Aniversario

Frederick P. Brook, Jr.

Universidad de Carolina del Norte en Chapel Hill

Traducción:
Aldo Nuñez Tovar
https://github.com/lizard20/El_Mitico_Hombre-Mes
anunez20@hotmail.com

Dedicatoria de la edición de 1975

A dos personas que de forma especial enriquecieron mi estancia en IBM:
Thomas J. Watson, Jr.,
cuya profunda preocupación por la gente aún permea su empresa,
y
Bob O. Evans,
cuyo audaz liderazgo convirtió el trabajo en aventura.

Dedicatoria de la edición de 1995

A Nancy,
fue un regalo de dios.

Prefacio a la 20ma. Edición de Aniversario

Estoy sorprendido y satisfecho pues *El Mítico Hombre-Mes* continúa siendo popular después de 20 años. Se han imprimido más de 250,000 copias. Con frecuencia me preguntan qué opiniones y recomendaciones expuestas en 1975 todavía sostengo, y cuáles han cambiado y cómo. De vez en cuando he abordado tales preguntas en conferencias, aunque desde hace tiempo he querido hacerlo por escrito.

Peter Gordon, ahora socio editorialista de Addison-Wesley, ha estado trabajando conmigo paciente y amablemente desde 1980. Me propuso preparar una Edición de Aniversario. Decidimos no revisar el original, sino reimprimirlo sin modificaciones (a excepción de algunas correcciones triviales) y añadirle reflexiones más actuales.

El capítulo 16 reimprime "No Existen Balas de Plata: Lo esencial y lo accidental de la Ingeniería de Software", un artículo publicado en 1986 para la confe-rencia del IFIP que surgió a raíz de mi experiencia cuando presidí una investigación del Consejo de Ciencia de la Defensa acerca del software militar. Mis coautores de ese estudio, y nuestro secretario ejecutivo, Robert L. Patrick, fueron invaluables al ponerme en contacto nuevamente con proyectos de software grandes del mundo real. El artículo fue reimprimido en 1987 en la revista *Computer magazine* de la IEEE, lo cual le dió una amplia difusión.

El artículo "No Existen Balas de Plata" demostró ser provocativo. Predijo que en una década no se observaría ninguna técnica de programación que trajera por sí misma una mejora de un orden de magnitud en la productividad

del software. Falta un año para la década; y mis predicciones parecen seguras. "No Existen Balas de Plata" ha estimulado mucho más el espíritu de discusión en la literatura que *El Mítico Hombre-Mes*. Por lo tanto, el Capítulo 17 habla acerca de algunas de las críticas publicadas y actualiza las opiniones expuestas en 1986.

Al preparar mi retrospectiva y actualización de *El Mítico Hombre-Mes*, me impresionó cuán pocas afirmaciones del libro han sido criticadas, demostradas, o refutadas por la actual investigación y experiencia de la ingeniería de software. Esto ahora me resulta útil para catalogar esas afirmaciones de forma cruda, desprovistas de hipótesis y datos de apoyo. Con la esperanza de que estas francas declaraciones inviten a debatir y a los hechos a demostrar, refutar, actualizar, o mejorar dichas proposiciones, las he incluido en el Capítulo 18 en forma de resumen.

El Capítulo 19 es en sí el ensayo de actualización. Se advierte al lector que estas nuevas opiniones no están ni de cerca tan bien respaldadas por la experiencia en las trincheras como en el libro original. He estado trabajando en una universidad, no en una industria, y en proyectos de pequeña escala, no en grandes. Desde 1986, solo he enseñado ingeniería de software, no he realizado investigación en este campo en absoluto. Mi investigación más bien se ha centrado en los ambientes virtuales y sus aplicaciones.

En la preparación de esta retrospectiva, he buscado las opiniones actualizadas de amigos que de hecho están trabajando en la ingeniería de software. Por la maravillosa buena voluntad de compartir sus puntos de vista, a comentar concienzudamente los borradores, y a reeducarme, estoy en deuda con Barry Boehm, Ken Brooks, Dick Case, James Coggins, Tom DeMarco, Jim McCarthy, David Parnas, Earl Wheeler, y Edward Yourdon. Fay Ward se ha encargado de forma magnífica de la produccion técnica de los nuevos capítulos.

Agradezco a Gordon Bell, Bruce Buchanan, Rick Hayes-Roth, mis colegas en la Fuerza de Tarea del Comité de Ciencia de la Defensa acerca del Software Militar, y, muy especialmente, a David Parnas por sus profundas y estimulantes ideas, y a Rebekah Bierly por la producción técnica del artículo imprimido aquí como Capítulo 16. Analizar el problema del software en categorías de *esencia* y *accidente* fue inspirado por Nancy Greeenwood Brooks, que utilizó dicho análisis en un artículo acerca de la pedagogía del violín Suzuki.

En el prefacio de la edición de 1975 los derechos de la casa editorial Addison Wesley no me permitieron reconocer los roles claves jugados por su personal. Debo citar especialmente las contribuciones de dos personas: Norman Stanton, Editor Ejecutivo en ese entonces, y Herber Boes, Director de Arte en aquella época. Boes diseñó el estilo elegante, fue especialmente mencionado por un revisor: "amplios márgenes, [y] un uso imaginativo de la tipografía y la plantilla." Y más importante aún, él también me hizo la recomendación crucial de que cada capítulo tuviera un dibujo inicial. (En ese entonces solo tenía el Pozo de Brea y la Catedral de Reims.) Encontrar los dibujos me ocasionó un año extra de trabajo, pero estoy eternamente agradecido por el consejo.

Soli Deo gloria– Gloria solo a dios

Chapel Hill, N.C. F. P. B., Jr.
Marzo de 1995

Prefacio a la Primera Edición

En muchos sentidos, la gestión de un proyecto grande de programación de sistemas es como gestionar cualquier otra gran empresa –en más formas de las que la mayoría de los programadores creen. Pero en muchos otros sentidos es diferente–en más formas de las que la mayoría de los gestores esperan.

El conocimiento del área se ha estado acumulando. Se han llevado a cabo varios congresos, sesiones en congresos de la AFIPS, y publicado algunos libros y artículos. Pero de ninguna manera aún en forma de un enfoque sistemático de libro de texto. Sin embargo, me parece apropiado ofrecer este pequeño libro, que refleja esencialmente un punto de vista personal.

Aunque originalmente me desarrollé en el lado de la programación, estuve principalmente involucrado en la arquitectura de hardware durante los años (1956-1963), en los que se desarrolló el programas de control autónomo y el compilador de lenguajes de alto nivel. Cuando en 1964 llegué a ser director del Operative System/360, encontré el mundo de la programación bastante cambiado debido al progreso de los últimos años.

Gestionar el desarrollo del OS/360 fue una experiencia muy educativa, aunque también algo muy frustrante. El equipo, incluyendo a F. M. Trapnell que me sucedió como director, tiene mucho de que enorgullecerse. El sistema contiene muchas virtudes en su diseño y ejecución, y ha tenido éxito en lograr un amplio uso. Ciertas ideas, las más evidentes son las entradas y salidas independientes del dispositivo y la gestión de bibliotecas externas, fueron innovaciones técnicas y que hoy en día son copiadas ampliamente. Actualmente

es bastante confiable, razonablemente eficiente y muy versátil.

Sin embargo, no puede decirse que el esfuerzo fue totalmente exitoso. Cualquier usuario del OS/360 se percata rápidamente de cuánto mejor debería ser. Los defectos en el diseño y la ejecución permearon especialmente al programa de control, a diferencia de los compiladores. La mayoría de estos defectos datan del periodo de diseño de 1964-65, por tanto son de mi responsabilidad. Además, el producto estaba retrasado, tomó más memoria de lo planificado, los costos superaron en mucho lo estimado, y no funcionó muy bien hasta varias entregas posteriores a la primera.

Luego de dejar IBM en 1965 para venir a Chapel Hill, como originalmente acordamos cuando asumí el cargo del OS/360, empecé a analizar la experiencia del OS/360 para ver qué lecciones técnicas y de gestión había que aprender. En particular, quería explicar las amplias diferencias en las experiencias de gestión encontradas en el desarrollo del hardware en la System/360 y el desarrollo del software del OS/360. Este libro es una respuesta tardía al cuestionario de Tom Watson de por qué la programación es difícil de gestionar.

En esta búsqueda me he beneficiado de largas conversaciones con R.P. Case, director adjunto de 1964-65, y con F. M. Trapnell, director de 1965-68. He comparado resultados con otros directores de proyectos grandes de programación, incluyendo a F.J. Corbató del M.I.T., a Jhon Harr y V. Vyssotsky de Bell Telephone Laboratories, a Charles Portman de International Computer Limited, a A. P. Ershov del Laboratorio de Computación de la División Siberiana, de la Academia de Ciencias de la U.R.S.S., y a A. M. Pietrasanta de IBM.

Mis propias conclusiones están plasmadas en los siguientes ensayos, y están dirigidas a los programadores profesionales, a los gestores profesionales, y especialmente a los gestores profesionales de la programación.

Aunque está escrito como ensayos separables, existe un tema principal contenido especialmente en los Capítulos del 2 al 7. En resumen, creo que los grandes proyectos de programación sufren de problemas de gestión de diferente tipo que los pequeños, a causa de la división del trabajo. Creo que la necesidad urgente es la preservación de la integridad conceptual del producto mismo. Estos capítulos exploran las dificultades por alcanzar esta unidad y los método para hacerlo. Los últimos capítulos exploran otros aspectos de la gestión de la ingeniería de software.

La literatura en este campo no es abundante, y está ampliamente dis-

persa. Por lo tanto, he tratado de dar referencias que por un lado aclaren puntos particulares y por otro guíen al lector interesado a otras obras útiles. Muchos amigos han leído el manuscrito, y algunos de ellos han preparado extensos comentarios provechosos; donde estos parecían valiosos pero no se ajustaban al flujo del texto, los incluía en las notas.

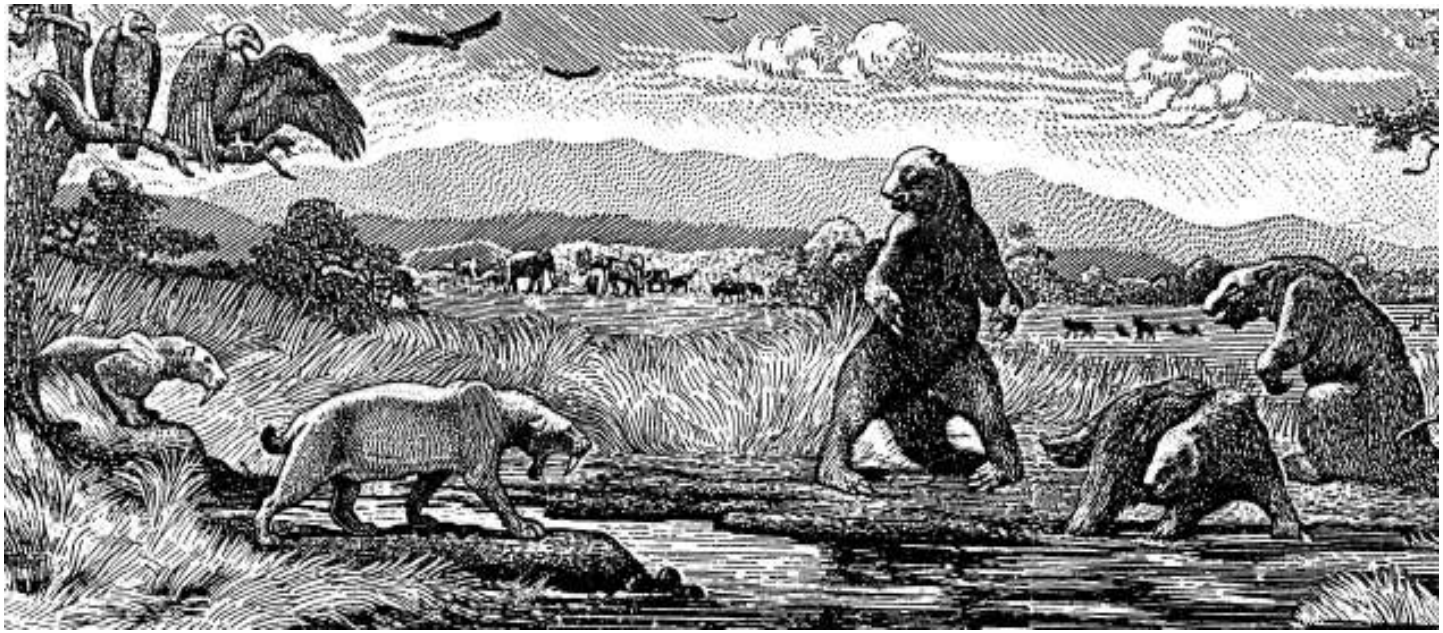
Debido a que este es un libro de ensayos y no uno de texto, todas las referencias y notas han sido desterradas al final del libro y se exhorta al lector a pasarlas por alto en una primera lectura.

Estoy profundamente en deuda con la Srita. Sara Elizabeth Moore, con el Sr. David Wagner, y con la Sra. Rebecca Burris por su ayuda en la preparación de este manuscrito, y con el Profesor Joseph C. Sloane por su consejo acerca de las ilustraciones.

Chapel Hill, N.C. F.P.B., Jr
Octubre de 1974

Contenido

1
El Pozo de Brea



1
El Pozo de Brea

Enn schip op het strand is een baken in zee.
[Un barco en la playa es un faro para la mar.]

PROVERVIO HOLANDÉS

C.R. Knight, Mural de La Brea Tar Pits
El Museo de George C. Page de La Brea Discoveries
Museo de Historia Natural del Condado de Los Ángeles

No hay escena de la prehistoria que sea tan intensa como aquella en donde grandes bestias luchan mortalmente en los pozos de brea. Uno se imagina a dinosaurios, mamuts, y tigres dientes de sable luchando en contra de la adhesión de la brea. Pero cuanto más ferozmente luchan, más los enreda la brea, y ninguna bestia es tan fuerte o tan hábil para evitar finalmente hundirse.

A lo largo de la década pasada la programación de sistemas grandes ha estado en tal pozo de brea, donde muchas bestias grandes y poderosas han sucumbido violentamente. A pesar de que la mayoría han emergido con sistemas que funcionan, unas pocas han cumplido con los objetivos, calendarios, y presupuestos. Grandes y pequeños, masivos o ligeros, equipo tras equipo se han enredado en la brea. Se puede apartar cualquier garra, pues nada parece causar el problema. Pero la acumulación de factores simultáneos e interactivos produce un movimiento cada vez más lento. Todos parecen estar sorprendidos por la persistencia del problema, y ha sido difícil discernir su naturaleza. Pero debemos intentar entender el problema si queremos resolverlo.

Por lo tanto, comencemos identificando el oficio de la programación de sistemas sus satisfacciones y sus infortunios inherentes.

El Producto de los Sistemas de Programación

Ocasionalmente leemos en las noticias relatos acerca de como dos programadores en un remodelado garaje han construido un programa importante que supera los mejores logros de los equipos grandes. Y todo programador está dispuesto a creer tales historias, pues sabe que podría construir *cualquier* programa mucho más rápido que las 1000 declaraciones/año reportadas por los equipos industriales.

¿Por qué entonces no se han reemplazado todos los equipos industriales por dedicados dúos de garaje? Echemos un vistazo a *eso* que se está produciendo.

En la parte superior izquierda de la Fig. 1.1 está un *programa*. Está completo en sí mismo, listo para ser ejecutado por el autor en el sistema en el que fue desarrollado. *Ese* es el objeto que comúnmente se produce en los garajes, y es el que utiliza el programador para evaluar la productividad.

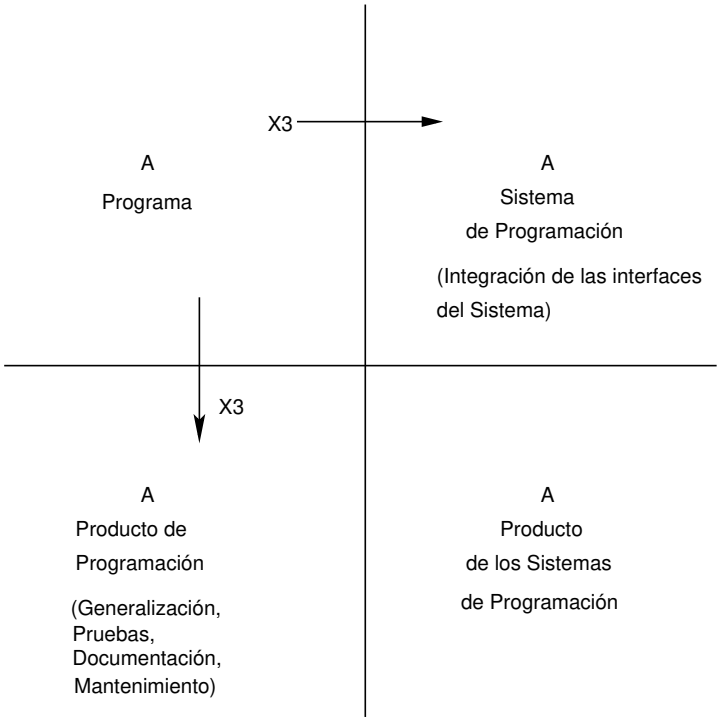


Fig. 1.1 Evolución del producto de los sistemas de programación

Hay dos formas en que un programa puede convertirse en un objeto más útil, pero más costoso. Estas dos formas están representadas por las fronteras del dibujo.

Si nos movemos hacia abajo y cruzamos la frontera horizontal, un programa llega a ser un *producto de programación*. Es decir, un programa que puede ser ejecutado, probado, reparado, y extendido por cualquiera. Está disponible para usarse en muchos ambientes operativos y para muchos conjuntos de datos. Para convertirse en un producto de programación que pueda usarse de forma general, un programa debe estar escrito en un estilo generalizado. En especial, la gama y la forma de las entradas deben ser tan generalizadas como razonablemente lo permita el algoritmo básico. Luego, el

programa debe ser probado minuciosamente, de modo que podamos confiar en él. Esto significa que se debe preparar, ejecutar y grabar un considerable banco de casos de prueba, que explore la gama de entradas y pruebe sus límites. Finalmente, el ascenso de un programa a un producto de programación requiere de una documentación minuciosa, tal que cualquiera pueda usarlo, corregirlo, y extenderlo. Como regla general, estimo que un producto de programación cuesta al menos tres veces más que un programa depurado con la misma funcionalidad.

Si nos movemos a través de la frontera vertical, un programa se convierte en un componente dentro de un *sistema de programación.* Esto es, en una colección de programas interactivos, cuyas actividades están coordinadas y cuyos formatos son estrictos, de tal modo que este ensamblaje constituya una instalación completa para tareas grandes. Un programa, para convertirse en un componente de un sistema de programación, debe escribirse de tal manera que toda entrada y toda salida se ajuste en sintaxis y semántica con interfaces muy bien definidas. El programa también debe diseñarse de tal manera que utilice solo un presupuesto determinado de recursos – espacio de memoria, dispositivos de entrada y salida, tiempo de cómputo. Por último, el programa debe probarse junto con otros componentes del sistema, en todas las combinaciones esperadas. Esta prueba debe ser amplia, por el número de casos que crece combinatoriamente. Todo esto consume tiempo, debido a los imperceptibles errores que surgen a partir de interacciones inesperadas entre componentes depurados. Un componente de un sistema de programación cuesta tres veces más que un programa autónomo con la misma funcionalidad. El costo puede ser mayor si el sistema tiene muchos componentes.

En la esquina inferior derecha de la Fig 1.1 se sitúa el *producto de los sistemas de programación.* Éste difiere del programa sencillo en todas las formas anteriores. Cuesta nueve veces más. Pero es el objeto verdaderamente útil, el producto deseado de la mayor parte de los esfuerzos de la programación de sistemas.

Las Satisfacciones del Oficio

¿Por qué la programación es divertida? ¿Qué satisfacciones pueden esperar sus practicantes como recompensa?

Primero, es la pura satisfacción de hacer cosas. Así como el niño se deleita en su pastel de lodo, así el adulto disfruta construyendo cosas, especialmente cosas de su propio diseño. Pienso que esta satisfacción debe ser una imagen de la satisfacción de dios al construir cosas, una satisfacción demostrada en la individualidad y originalidad de cada hoja y cada copo de nieve.

Segundo, es el placer de hacer cosas que sean útiles para otras personas. En el fondo, queremos que otros usen nuestro trabajo y lo encuentren útil. En este sentido, la programación de sistemas no es esencialmente diferente al primer portalápices de arcilla del niño "para la oficina de papá".

Tercero, es la fascinación de crear objetos complejos, como rompecabezas, de piezas móviles entrelazadas y observarlos trabajar en ciclos sutiles, agotando los efectos de los principios incorporados desde el inicio. La computadora programada tiene toda la fascinación de la máquina de pinball o del mecanismo de la rocola, llevada al extremo.

Cuarto, es la satisfacción del continuo aprendizaje, que brota de la naturaleza no repetitiva del trabajo. De una forma u otra el problema es siempre nuevo, y quien lo resuelve aprende algo: algunas veces algo práctico, otras algo teórico y a veces ambos.

Finalmente, existe la satisfacción de trabajar en un medio tan manipulable. El programador, como el poeta, trabaja sólo ligeramente alejado de ese instrumento puramente intelectual. Construye sus castillos en el aire, a partir del aire, crea a través del esfuerzo de la imaginación. Pocos medios de la creación son tan flexibles, tan fáciles de pulir y reelaborar, tan fácilmente capaces de llevar a cabo estupendas estructuras conceptuales. (Como veremos después, esta gran docilidad tiene sus propios problemas.)

Sin embargo, la construcción de programas a diferencia de las palabras del poeta, es real en el sentido que se mueve y funciona, y produce salidas visibles separadas de la construcción misma. Imprime resultados, dibuja, produce sonidos, mueve brazos. La magia del mito y la leyenda se ha hecho realidad en nuestro tiempo. Uno teclea el conjuro correcto, y la pantalla cobra vida, mostrando cosas que jamás fueron ni podrían ser.

Por lo tanto, la programación es divertida porque gratifica el anhelo cre-

ativo inmerso en lo profundo de nosotros y deleita los sentimientos que tenemos en común con todas las personas.

Los Infortunios del Oficio

Sin embargo, no todo es satisfacción y conocer los infortunios intrínsecos nos ayudará para lidiar con ellos cuando surjan.

En primer lugar, se debe actuar perfectamente. La computadora también se parece a la magia de la leyenda en este aspecto. Si un personaje o una pausa del conjuro no está estrictamente en la forma apropiada, la magia no funciona. Los seres humanos no estamos acostumbrados a ser perfectos, y pocas áreas de la actividad humana lo exigen. Pienso que ajustarse al requisito de perfección es la parte más difícil del aprendizaje de la programación.[1]

Luego, otras personas nos asignan los objetivos, nos proveen los recursos, y nos facilitan la información. Uno rara vez controla las circunstancias de su trabajo, o incluso su objetivo. En términos de gestión, nuestra autoridad no es suficiente por toda nuestra responsabilidad. Sin embargo, parece que en todos los trabajos donde se hacen cosas no se tiene una autoridad formal acorde a la responsabilidad. En la práctica, la autoridad real (como opuesta a la formal) se adquiere del mismo impulso del cumplimiento.

La dependencia de otros tiene una particularidad especialmente terrible para el programador de sistemas. Pues depende de los programas de otras personas. Y estos a menudo están mal diseñados, mal implementados, liberados de forma incompleta (sin código fuente ni casos de prueba), y mal documentados. De tal manera que se debe invertir horas estudiando y corrigiendo cosas que en un mundo ideal deberían estar completas, disponibles y listas para usarse.

El siguiente infortunio es que diseñar conceptos estupendos es divertido; encontrar una plaga de pequeños errores es solo trabajo. Acompañada de cualquier actividad creativa vienen monótonas horas de trabajo tedioso, meticuloso, y la programación no es la excepción.

Luego, uno descubre que la depuración tiene una convergencia lineal o peor, donde uno esperaba, por algún motivo, un tipo de convergencia cuadrática hacia el final. De este modo, las pruebas se hacen interminables, encontrar los últimos errores difíciles toma más tiempo que encontrar los primeros.

El último infortunio, y a veces la gota que colma el vaso, es que el producto en el que se ha trabajado durante tanto tiempo parece estar quedando obsoleto al finalizar (o antes). Los colegas y competidores ya están en la intensa búsqueda de nuevas y mejores ideas. La sustitución del pensamiento infantil ya no solo está concebida, sino programada.

Esto siempre parece peor de lo que realmente es. El producto nuevo y mejorado generalmente no está *disponible* cuando uno termina el propio; de esto solo se habla. Y esto también requerirá meses de desarrollo. Jamás equiparemos un tigre de verdad con uno de papel, salvo que queramos usarlo de verdad. Por lo tanto, las ventajas de la realidad se satisfacen por sí mismas.

Por supuesto que la base tecnológica sobre la cual uno construye avanza *siempre*. Tan pronto como se congela el diseño, se vuelve obsoleto en términos de sus conceptos. Pero la implementación de productos reales exige sincronización y cuantificación. La obsolescencia de una implementación se debe medir contra otras implementaciones existentes, no contra conceptos no realizados. El reto y la misión son encontrar soluciones reales a problemas reales en base a calendarios reales de acuerdo a los recursos disponibles.

Pues esto es la programación, por un lado el pozo de brea, en el cual muchos esfuerzos han sucumbido, y por otro, la actividad creativa con sus propias satisfacciones e infortunios. Para muchos, las satisfacciones superan con creces a los infortunios, y para ellos el resto de este libro intentará tender algunas vías a través de la brea.

2
El Mítico Hombre-Mes

Restaurant Antoine

Fondé En 1840

AVIS AU PUBLIC

Faire de la bonne cuisine demande un certain temps. Si on vous fait attendre, c'est pour mieux vous servir, et vous plaire.

ENTREES (SUITE)

Côtelettes d'agneau grillées 2.50
Côtelettes d'agneau aux champignons frais 2.75
Filet de boeuf aux champignons frais 4.75
Ris de veau à la financière 2.00
Filet de boeuf nature 3.75
Tournedos Médicis 3.25
Pigeonneaux sauce paradis 3.50
Tournedos sauce béarnaise 3.25
Entrecôte minute 2.75
Filet de boeuf béarnaise 4.00
Tripes à la mode de Caen (commander d'avance) 2.00

Entrecôte marchand de vin 4.00
Côtelettes d'agneau maison d'or
Côtelettes d'agneau à la parisienne
Foie de volaille à la brochette 1.50
Tournedos nature 2.75
Filet de boeuf à la hawaïenne 4.00
Tournedos à la hawaïenne 3.25
Tournedos marchand de vin 3.25
Pigeonneaux grillés 3.00
Entrecôte nature 3.75
Châteaubriand (30 minutes)

LÉGUMES

Epinards sauce crême .60
Broccoli sauce hollandaise .80
Pommes de terre au gratin .60
Haricots verts au beurre .60
Chou-fleur au gratin .60
Asperges fraîches au beurre .90
Carottes à la crème .60
Pommes de terre soufflées
Petits pois à la française .75

SALADES

Salade Antoine .60
Salade Mirabeau .75
Salade laitue au roquefort .80
Salade de laitue aux tomates .60
Salade de légumes .60
Salade d'anchois 1.00
Fonds d artichauts Bayard
Salade de laitue aux oeufs .60
Tomate frappée à la Jules César .60
Salade de coeur de palmier 1.00
Salade aux pointes d'asperges .60
Avocat à la vinaigrette .60

DESSERTS

Gâteau moka .50
Méringue glacée .60
Crêpes Suzette 1.25
Glace sauce chocolat .60
Fruits de saison à l'eau-de-vie .75
Omelette soufflée à la Jules César (2) 2.00
Omelette Alaska Antoine (2) 2.50
Cerises jubilé 1.25
Crêpes à la gelée .80
Crêpes nature .70
Omelette au rhum 1.10
Glace à la vanille
Fraises au kirsch
Pêche Melba

FROMAGES

Roquefort .50
Camembert .50
Liederkranz .50
Gruyère .50
Fromage à la crême Philadelphie .50

CAFÉ ET THÉ

Café .20
Café brulôt diabolique 1.00
Café au lait .20
Thé glacé .20
Thé .20
Demi-tasse

EAUX MINERALES—BIERE—CIGARES—CIGARETTES

White Rock
Vichy
Cliquot Club
Bière locale
Canada Dry
Cigarettes

Roy b. Alciatore, Propriétaire

713-717 Rue St. louis Nouvelle Orléans, louisiane

2
El Mítico Hombre-Mes

La buena cocina toma tiempo. Si usted está dispuesto a esperar, es para servirle mejor y complacerlo.

MENÚ DEL RESTAURANTE ANTOINE. NEW ORLEANS

Más proyectos de software han fracasado por la escasez de tiempo de calendario que debido a la combinación de otras causas. ¿Por qué la causa más común de este desastre resulta ser el calendario?

Primero, nuestras técnicas de estimación están poco desarrolladas. Y más grave aún es que reflejan una suposición que no se menciona y que es bastante falsa, i.e., que todo saldrá bien.

Segundo, nuestras técnicas de estimación confunden de forma equivocada esfuerzo con progreso, y esconden esa suposición de que hombres y meses son intercambiables.

Tercero, debido a la incertidumbre de nuestras estimaciones, los gestores de software carecen de la amable tenacidad del cocinero del Antoine.

Cuarto, el calendario de avances está mal supervisado. En la ingeniería de software se consideran innovaciones radicales a técnicas comprobadas y habituales en otras ramas de la ingeniería.

Quinto, cuando se detecta un retraso en el calendario, la respuesta normal (y típica) es añadir mano de obra. Esto es como apagar el fuego con gasolina, empeora mucho más las cosas. Más fuego exige más gasolina, y así se inicia un ciclo regenerativo con final desastroso.

La supervisión del calendario será materia de un ensayo aparte. Consideremos otros aspectos del problema con mayor detalle.

Optimismo

Todos los programadores son optimistas. Quizá sea porque este embrujo moderno atrae en especial a aquellos que creen en finales felices y hadas madrinas. O quizá tantas pequeñas decepciones ahuyentan a todos excepto a aquellos que habitualmente se concentran en el objetivo final. O a lo mejor es simplemente porque las computadoras son jóvenes, los programadores son más jóvenes, y los jóvenes son siempre optimistas. Más sin embargo, el proceso de selección funciona, el resultado es irrefutable: "Esta vez con seguridad funcionará," o "Acabo de encontrar el último error."

Así pues, la primera falsa suposición que subyace al calendario de la programación de sistemas es que *todo saldrá bien*, i.e., que *cada tarea tardará solo el tiempo que "deba" hacerlo.*

La omnipresencia del optimismo entre los programadores merece más que un simple análisis. Dorothy Sayers, en su excelente libro, *La Mente del*

Creador, divide la actividad creativa en tres etapas: la idea, la implementación, y la interacción. Así, un libro, o una computadora, o un programa llega a existir primero como una construcción ideal, elaborada fuera del tiempo y del espacio, aunque íntegra en la mente del autor. Se produce en el tiempo y el espacio, con pluma, tinta y papel, o a través de alambres, silicio y ferrita. La creación concluye cuando alguien lee el libro, usa la computadora, o ejecuta el programa, interactuando así con la mente del creador.

Esta descripción, que la Sra. Sayers usa para iluminar no solo el quehacer creativo sino también la doctrina cristiana de la trinidad, nos ayudará en la presente tarea. Para el ser humano hacedor de cosas, las incompletitudes e inconsistencias de nuestras ideas solo se aclaran durante la implementación. Así es como la escritura, la experimentación y "la elaboración" son prácticas esenciales para el teórico.

En muchos trabajos creativos el medio de ejecución es inmanejable. Maderas partidas; manchas de pintura; corto circuitos eléctricos. Estas limitaciones físicas del medio restringen la expresión de las ideas, y además crean dificultades inesperadas a la hora de la implementación.

Así pues, la implementación toma tiempo y sudor debido tanto al medio físico como por lo inadecuado de nuestras ideas subyacentes. Tendemos a culpar a los medios físicos por la mayoría de nuestras dificultades en la implementación; porque los medios no son "nuestros" en la forma en que lo son las ideas, además nuestro orgullo tiende a empañar nuestro juicio.

Sin embargo, la programación de computadoras crea con un medio excesivamente maleable. El programador construye a partir de un instrumento puramente intelectual: los conceptos y sus tan fexibles representaciones. Debido a que el medio es maleable, esperamos pocas dificultades en la implementación; de ahí nuestro optimismo omnipresente. Debido a que nuestras ideas son defectuosas, cometemos errores; de ahí que nuestro optimismo sea injustificado.

En una sola tarea, la suposición de que todo saldrá bien tiene un efecto probabilístico en el calendario. Y efectivamente, la tarea podría ir según lo planeado, pues descubriremos que existe una distribución de probabilidad para el retraso, y el "no retraso" tiene una probabilidad finita. Sin embargo, un proyecto de programación grande consta de muchas tareas, algunas encadenadas de extremo a extremo. La probabilidad de que cada una salga bien

será cada vez más pequeña.

El Hombre-Mes

El segundo modo de pensamiento falaz se expresa en la propia unidad de trabajo utilizada en la estimación y en la calendarización: el hombre-mes. El costo, en efecto, varía como el producto del número de hombres y el número de meses. El progreso no. *Por lo tanto, el hombre-mes como unidad de medida del tamaño de un trabajo es un mito peligroso y engañoso.* Pues implica que los hombres y los meses son intercambiables.

Los hombres y los meses son productos intercambiables solo cuando una tarea puede dividirse entre muchos trabajadores *sin comunicación entre ellos* (Fig. 2.1). Esto es cierto para la cosecha de trigo o la recolección de algodón; pero no es ni siquiera aproximadamente cierto en la programación de sistemas.

Fig. 2.1 Tiempo versus número de trabajadores – una tarea perfectamente divisible

Cuando una tarea no puede dividirse debido a restricciones secuenciales, la aplicación de un mayor esfuerzo no tiene efecto sobre el calendario (Fig. 2.2). Dar a luz a un niño tarda nueve meses, sin importar cuantas mujeres se asignen. Muchas tareas de software tienen esta característica debido a la naturaleza secuencial de la depuración.

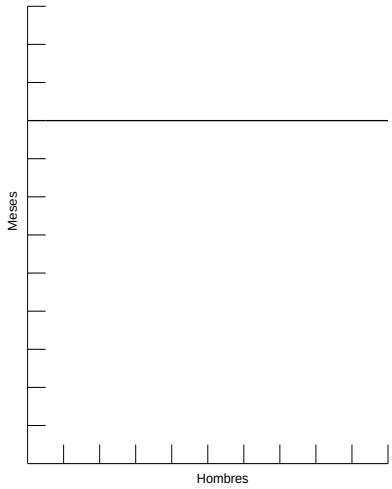


Fig. 2.2 Tiempo versus número de trabajadores – una tarea no divisible

En tareas que pueden dividirse pero que requieren comunicación entre sus subtareas, hay que añadir a la cantidad total de trabajo el esfuerzo de comunicación. Por lo tanto, lo mejor que podemos obtener es algo más pobre que un intercambio equitativo de hombres por meses (Fig. 2.3).

Fig. 2.3 Tiempo versus número de trabajadores – una tarea divisible que requiere comunicación

La carga adicional de la comunicación está compuesta de dos partes, la capacitación y la intercomunicación. Cada trabajador debe capacitarse en la tecnología, las metas del esfuerzo, la estrategia global y el plan de trabajo. Esta capacitación puede dividirse, así que esta parte del esfuerzo adicional varía linealmente con el número de trabajadores.[1]

La intercomunicación es peor. Si cada parte de la tarea debe coordinarse de forma separada entre ellas, el esfuerzo se incrementa como $n(n-1)/2$. Tres trabajadores requieren tres veces más intercomunicación por parejas que dos; cuatro trabajadores requieren seis veces más que dos. Si además, se requieren reuniones de trabajo entre tres, cuatro, etc., trabajadores para resolver cosas de manera conjunta, las cosas empeoran aún más. El esfuerzo adicional de comunicación puede contrarrestar totalmente la división de la tarea original y llevarnos a la situación de la Fig. 2.4.

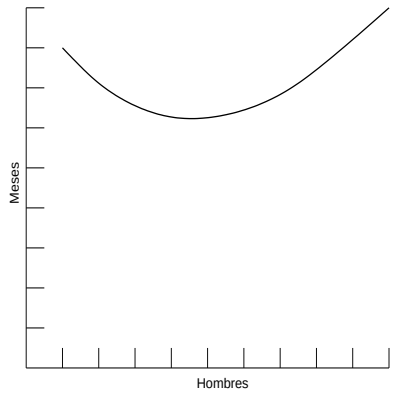


Fig. 2.4 Tiempo versus número de trabajadores – una tarea con interrelaciones complejas

Puesto que la construcción del software es intrínsecamente un esfuerzo de sistemas – una actividad de interrelaciones complejas – el esfuerzo de comunicación es grande, y domina rápidamente la reducción de tiempo de cada tarea que conlleva la división. Por lo tanto, añadir más personas no reduce sino extiende el calendario.

Pruebas de Sistemas

Ninguna parte del calendario se ve más afectada por las restricciones secuenciales que la depuración de componentes y la prueba del sistema. Además, el tiempo requerido depende de la cantidad y lo escurridizo de los errores hallados. Teóricamente este número debería ser cero. En virtud de nuestro optimismo, generalmente esperamos que el número de errores sea menor de

lo que finalmente resulta ser. Por lo tanto, las pruebas generalmente representan la parte peor calendarizada de la programación.

Por algunos años he estado utilizando con éxito la siguiente regla empírica para calendarizar una tarea de software.

1/3 planificación
1/6 codificación
1/4 prueba de componentes y primera prueba del sistema
1/4 pruebas del sistema, todos los componentes disponibles

Esta regla difiere de las calendarizaciones habituales en varios aspectos importantes:

1. La fracción dedicada a la planificación es mayor de lo normal. Aun así, apenas es suficiente para realizar una especificación sólida y detallada, y no es suficiente para incluir la exploración o la investigación de técnicas totalmente nuevas.
2. La *mitad* del calendario dedicada a la depuración de programas terminados es mucho mayor de lo normal.
3. A la parte que es fácil de estimar, i.e., la codificación, se le asigna solo una sexta parte del calendario.

Al examinar proyectos calendarizados de forma tradicional, encontré que pocos asignaban una mitad del calendario a pruebas, aunque la mayoría en efecto invertía la mitad del calendario real para este propósito. Muchos de ellos cumplían con el calendario hasta y exceptuando en la prueba del sistema.[2]

En particular, errar en asignar tiempo suficiente a la prueba del sistema es especialmente desastroso. Puesto que los retrasos vienen al final del calendario, nadie está consciente de los problemas hasta casi la fecha de entrega. Malas noticias, retrasado y sin previo aviso, esto es preocupante para clientes y gestores.

Además, el retraso en esta etapa tiene repercusiones financieras inusualmente severas, como también psicológicas. El personal del proyecto está totalmente completo, y el costo por día es máximo. Más grave aún, el software es para apoyar otros empredimientos comerciales (el envío de computadoras, la operación de nuevas instalaciones, etc.) y los costos secundarios de retrasarlos son muy altos, pues ya casi es hora del envío del software. Y efectivamente, estos costos secundarios pueden superar con creces a todos los demás. Por lo tanto, es muy importante permitir tiempo suficiente a la prueba

del sistema en el calendario original.

Estimación sin Agallas

Observe que para el programador, como para el cocinero, la urgencia del cliente puede dictar la conclusión de la tarea, pero no puede dictar su conclusión real. Una tortilla, prometida en dos minutos, puede dar la impresion de estar marchando bien. Pero cuando no ha cuajado en dos minutos, el cliente tiene dos opciones – esperar o comerlo crudo. Los clientes de software tienen las mismas opciones.

El cocinero tiene todavía otra opción más; puede aumentar el fuego. El resultado con frecuencia es una tortilla que nada puede salvar – quemada en una parte, y cruda en otra.

Ahora bien, no creo que los gestores de software tengan menos coraje y firmeza inherentes que los cocineros, ni que otros gestores de la ingeniería. Pero el falso calendario con tal de cumplir con la fecha que el cliente desea es mucho más común en nuestra disciplina que en otro lugar de la ingeniería. Es muy difícil hacer una defensa vigorosa, convincente y con el riesgo de perder el trabajo de una estimación obtenida a través de un método no cuantitativo, respaldada por pocos datos, y apoyada principalmente por las intuiciones de los gestores.

Claramente necesitamos dos soluciones. Tenemos que desarrollar y difundir cifras de productividad, cifras de incidencia de errores, reglas de estimación, etcétera. La profesión entera solo puede beneficiarse compartiendo tales datos.

Hasta que la estimación se asiente sobre bases sólidas, los gestores deben reforzar sus agallas y defender sus estimaciones con la seguridad de que sus modestas intuiciones son mejores que las estimaciones derivadas de sus deseos.

Desastre Regenerativo del Calendario

¿Qué hacer cuando un proyecto de software fundamental está retrasado? Añadir mano de obra, naturalmente. Como sugieren las figuras de la 2.1 a la 2.4, esto puede ayudar o no.

Veamos un ejemplo.[3] Supongamos que una tarea está estimada en 12 hombres-mes y se asignan tres personas por cuatro meses, y además existen hitos medibles A, B, C, D, que están calendarizados para caer al final de cada mes (Fig. 2.5).

Ahora supongamos que no se cumplió el primer hito hasta transcurridos dos meses (Fig. 2.6). ¿Qué opciones enfrenta el gestor?

1. Supongamos que la tarea debe terminarse a tiempo. Suponer que solo la primera parte de la tarea fue mal estimada, es la historia que cuenta exactamente la Fig. 2.6. Entonces, restan 9 hombres-mes de trabajo y dos meses, así que se necesitarán $4\frac{1}{2}$ personas. Hay que añadir 2 personas a las 3 ya asignadas.

2. Supongamos que la tarea debe terminarse a tiempo. Suponer que toda la estimación fue mala en general, la Fig. 2.7 describe realmente esta situación. Por lo tanto, restan 18 hombres-mes de trabajo y dos meses, así que se necesitarán 9 personas. Hay que añadir 6 personas a las 3 ya asignadas.

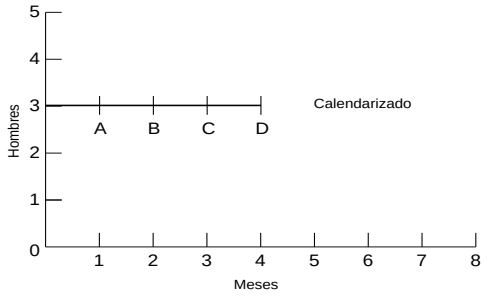


Figura 2.5

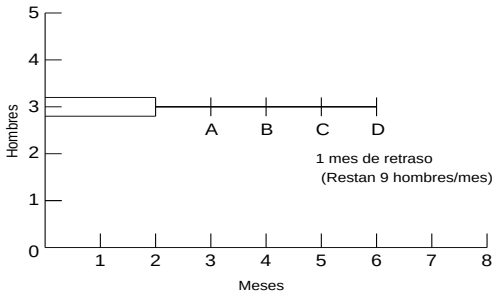


Figura 2.6

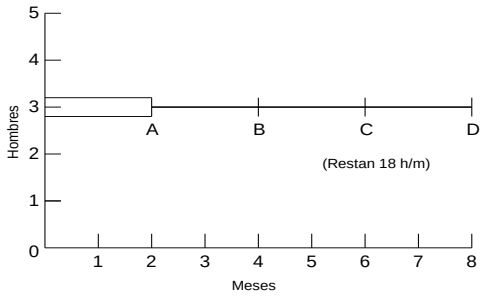


Figura 2.7

3. Recalendarice. Me gusta el consejo dado por P. Fagg, un experimentado ingeniero de hardware, "No cometa tropiezos pequeños." Es decir, permitir tiempo suficiente en el nuevo calendario para garantizar que el trabajo se lleve a cabo cuidadosa y completamente, y no se tenga que recalendarizar otra vez.

4. Recorte la tarea. En la práctica esto tiende a suceder de cualquier manera, una vez que el equipo observa retrasos en el calendario. Esta es la única acción viable cuando los costos secundarios por el retraso son muy elevados. Las únicas alternativas del gestor son: recortar la tarea formal y cuidadosamente, para recalendarizar, u observar que la tarea se recorte silenciosamente mediante un diseño precipitado y pruebas incompletas.

En los dos primeros casos, insistir en que la tarea inalterada sea concluida en cuatro meses es desastroso. Por ejemplo, considere los efectos regenerativos en la primera alternativa (Fig. 2.8). Las dos personas nuevas, independientemente de ser competentes y haber sido reclutadas a prisa, requerirán capacitación en la tarea por parte de una persona con experiencia. Si esto toma un mes, *se habrán dedicado 3 hombres-mes a trabajar pero no en la estimación original*. Además, la tarea dividida originalmente en tres partes, debe dividirse nuevamente en cinco partes: por lo tanto, se perderá algo del trabajo ya realizado, y las pruebas del sistema deberán ser prolongadas. Así, al final del tercer mes, restan en esencia más de 7 hombres-mes de trabajo, y se dispone de 5 personas capacitadas y un mes. Como la Fig. 2.8 sugiere, el producto está tan retrasado como si no se hubiera agregado a nadie (Fig. 2.6).

El deseo de llevarlo a cabo en cuatro meses, considerando solo el tiempo de capacitación sin repartición ni pruebas extras de los sistemas, requeriría añadir al final del segundo mes, 4 personas, no 2. Para cubrir los efectos de la repartición y la prueba del sistema, se tendría que añadir otra persona más. Sin embargo, ahora al menos tenemos un equipo de 7 personas, no uno de 3; así pues, tales aspectos como la organización y la división de tareas son diferentes en tipo, no solo en grado.

Observe que al final de tercer mes el panorama es sombrío. El hito del 1 de Marzo no ha sido alcanzado a pesar de todo el trabajo de gestión. La tentación

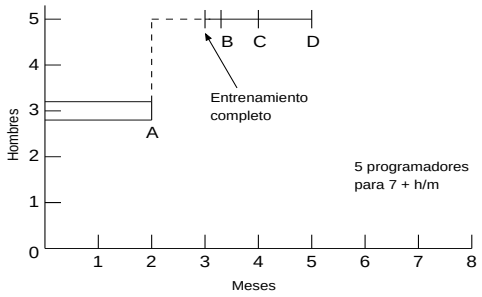


Figura 2.8

de repetir el ciclo es muy fuerte, añadir aún más mano de obra. Ahí yace la locura.

Lo anterior supuso que solo el primer hito fue mal estimado. Si el 1 de Marzo se hace la suposición conservadora de que todo el calendario fue optimista, como muestra la Fig . 2.7, uno busca añadir 6 personas solo a la tarea original. El cálculo de los efectos de la capacitación, la repartición y la prueba del sistema se deja al lector como ejercicio. Sin duda alguna, el desastre regenerativo conducirá a un producto más pobre, más retrasado, que la recalendarización con las tres personas originales, sin añadir a nadie.

Sobresimplificando exageradamente, enunciamos la Ley de Brooks:

> *Añadir mano de obra a un proyecto de software retrasado lo retrasa aún más.*

Pues esta es la desmitificación del hombre-mes. El número de meses de un proyecto está sujeto a restricciones secuenciales. El máximo número de personas está sujeto al número de subtareas independientes. A partir de estas dos cantidades podemos obtener calendarios que usen menos personas y más meses. (El único riesgo es la obsolescencia del producto.) Sin embargo, no se

pueden obtener calendarios viables con más personas y menos meses. Gran parte de los proyectos de software han fracasado más por la escasez de tiempo de calendario que debido a la combinación de otras causas.

3
El Equipo Quirúrgico

3
El Equipo Quirúrgico

Estos estudios han revelado que las diferencias entre individuos de alto y bajo rendimiento, generalmente son de un orden de magnitud.

SACKMAN, ERIKSON, Y GRANT[1]

Foto UPI/El Archivo Bettman

mentes En reuniones de la sociedad de computación, con frecuencia escucho a jóvenes gestores de programación afirmar que están a favor de un equipo pequeño e inteligente; de personal de primera clase, en lugar de un proyecto con cientos de programadores, y por ende mediocres. Nosotros también estamos a favor.

Aunque esta ingenua exposición de alternativas evade el meollo del problema – ¿cómo construir *sistemas grandes* en base a calendarios sensatos? Veamos cada lado de esta pregunta con mayor detalle.

El Problema

Hace mucho que los gestores de programación han reconocido la amplia variación entre los buenos y los malos programadores. Aunque nos ha asombrado a todos la magnitud de la medición real. En uno de sus estudios, Sackman, Erikson, y Grant, midieron el rendimiento de un grupo de experimentados programadores. Solo dentro de este grupo, las proporciones entre los mejores y peores rendimientos promediaban cerca de 10:1 en mediciones de productividad y un impresionante 5:1 en mediciones de velocidad y tamaño del programa! En resumen, el programador de 20,000 dólares/año bien podría ser 10 veces más productivo que uno de 10,000 dólares/año. Lo contrario también puede ser cierto. Los datos no mostraron ninguna correlación en absoluto entre experiencia y rendimiento. (Dudo que eso sea universalmente válido.)

Anteriormente he afirmado que la coordinación total del número de mentes afecta el costo del trabajo, ya que una parte importante del costo es la comunicación y la corrección de los efectos adversos de la mala comunicación (depuración del sistema). Esto, también, sugiere que uno busca que el sistema sea construido por el menor número de mentes posible. Y en efecto, gran parte de la experiencia con sistemas de programación grandes muestra que el enfoque de la fuerza bruta es costoso, lento, ineficiente y produce sistemas que no están conceptualmente integrados. OS/360, Exec 8, Scope 6600, Multics, TSS, SAGE, etc. – y la lista continúa.

La conclusión es simple: si un proyecto de 200 personas tiene 25 gestores que son los programadores más competentes y experimentados, despida a la tropa de 175 y ponga a los gestores de nuevo a programar.

Ahora examinemos esta solución. Por un lado, fracasa en el enfoque ideal de un equipo inteligente y *pequeño*, el cual por consenso no debe exceder las 10 personas. Este equipo como es muy grande necesitará al menos dos niveles de gestión, o cerca de cinco gestores. Y adicionalmente necesitará apoyo financiero, personal, espacio, secretarias y operadores de computadoras.

Por otro lado, el equipo original de 200 personas no era lo suficientemente grande como para construir sistemas realmente grandes a través de los métodos de la fuerza bruta. Considere el OS/360, por ejemplo. En el punto más alto trabajaban más de 1000 personas en él – programadores, escritores, operadores, archivistas, secretarias, gestores, grupos de apoyo y demás. Probablemente de 1963 a 1966 se dedicaron 5000 hombres-año a su diseño, construcción y documentación. A nuestro equipo candidato de 200 personas, le tomaría 25 años llevar el producto a su etapa actual, si los hombres y los meses fueran todavía intercambiables!

Pues este es el problema con el concepto de equipo pequeño e inteligente: *es muy lento para sistemas realmente grandes*. Considere el trabajo del OS/360 como si fuera a ser abordado por un equipo pequeño e inteligente. Escoja un equipo de 10-personas. Como límite, ya que son inteligentes, permita que sean siete veces más productivos que los programadores mediocres, respecto a la programación y la documentación. Suponga que el OS/360 fue construido por programadores mediocres (lo cual está *lejos* de ser cierto). Suponga además, como límite otro factor de siete en el aumento de la productividad que viene de la reducción en la comunicación por parte del equipo pequeño. Asuma que el *mismo* equipo permanece a lo largo de todo el proyecto. Ahora bien, $5000/(10 \times 7 \times 7) = 10$, ellos podrán realizar el trabajo de 5000 hombres-año en 10 años. ¿Será un producto interesante 10 años después de su diseño inicial? ¿O se habrá hecho obsoleto debido al acelerado desarrollo de la tecnología del software?

El dilema es cruel. Por eficiencia e integridad conceptual, es preferible que unas pocas mentes realicen el diseño y la construcción. Sin embargo, para los sistemas grandes se busca añadir considerable mano de obra de apoyo, tal que el producto se pueda presentar a tiempo. ¿Cómo podemos reconciliar estos dos requisitos?

La Propuesta de Mills

La propuesta de Harlan Mills ofrece una solución nueva y creativa.[2,3] Él propone que cada segmento de un trabajo grande sea abordado por un equipo, y que cada equipo sea organizado como un equipo quirúrgico en vez de un equipo de carnicería. Es decir, en lugar de que cada miembro haga cortes, solo uno hace los cortes y los demás le brindan apoyo para que mejore su eficacia y su productividad.

Si reflexionamos un poco vemos que este concepto cumple con los propósitos, si podemos lograr que funcione. Pocas mentes se involucran en el diseño y la construcción, aunque se usan muchas manos para ponerlo en marcha. ¿Funcionará? ¿Quiénes serán los anestesiólogos? y ¿Quiénes serán las enfermeras en el equipo de programación? y ¿Cómo se dividirá el trabajo? Permítanme mezclar metáforas libremente y sugerir cómo tal equipo podría funcionar si lo extendemos para incluir todo el apoyo posible.

El cirujano. Mills lo llama *programador líder*. Él personalmente define las especificaciones tanto funcionales como de rendimiento, diseña los programas, los codifica, los prueba, y escribe su documentación. Escribe en un lenguaje de programación estructurado como PL/I, y tiene un acceso eficaz al sistema de cómputo que no solo corre sus pruebas sino también almacena varias versiones de sus programas, permitiendo una fácil actualización de archivos, y le provee un editor de texto para que lleve a cabo la documentación. El cirujano debe tener un gran talento, diez años de experiencia, y un considerable conocimiento de sistemas y aplicaciones, ya sea en matemáticas aplicadas, en manejo de datos de negocios o algo semejante.

El copiloto. Es el alter ego del cirujano, es capaz de realizar cualquier parte del trabajo, aunque es menos experimentado. Su función principal es participar en el diseño como pensador, debatidor y evaluador. El cirujano le sugiere ideas, aunque no está sujeto a sus consejos. Normalmente, el copiloto representa a su equipo en discusiones con otros equipos acerca de la funcionalidad y las interfaces. Conoce todo el código íntimamente. Investiga estrategias alternativas de diseño. Obviamente sirve como seguro del cirujano contra desastres. Puede incluso escribir código, aunque no es responsable de ninguna parte del mismo.

El administrador. El cirujano es el jefe, y debe tener la última palabra acerca del personal, aumentos, espacio y demás, aunque casi no debe invertir su tiempo en estas cuestiones. Por lo tanto, necesita un administrador profesional que maneje el dinero, el personal, el espacio y las máquinas, y que interactúe con la maquinaria administrativa del resto de la organización. Baker sugiere que el administrador tenga un trabajo de tiempo completo solo si el proyecto tiene considerables requisitos legales, contractuales, de informes o financieros debido a la relación usuario-productor. De otra manera, un solo administrador puede servir a dos equipos.

El editor. El cirujano es el responsable de generar la documentación – para mayor claridad es necesario que él la escriba. Esto se cumple tanto para las descripciones internas como para las externas. Sin embargo, el editor toma el borrador o el manuscrito dictado por el cirujano y lo analiza, lo reelabora, añade referencias y bibliografía, lo mantiene a través de sus distintas versiones, y supervisa el proceso de producción.

Dos secretarias. El administrador y el editor necesitarán cada uno una secretaria: la secretaria administrativa manejará la correspondencia del proyecto y los archivos que no sean técnicos.

El archivista de programas. Es el responsable de mantenener todos los registros técnicos del equipo en una biblioteca de productos de programación. El archivista está entrenado como una secretario y es el responsable de los archivos legibles tanto por las computadoras como por los humanos.

Todo ingreso a la computadora va al archivista, quien lo registra y teclea en caso necesario. Asimismo, archiva e indexa los listados de salida. Las ejecuciones más recientes de cualquier modelo se mantienen en un cuaderno de estado; mientras que todas las anteriores se almacenan en un archivo cronológico.

Algo absolutamente vital al concepto de Mills es la transformación de la programación “de un arte privado a una práctica pública” pues logra visivilizar *todas* las operaciones de la computadora a todos los miembros del equipo e identifica a todos los programas y datos como propiedad del equipo,

no como propiedad privada.

La función especializada del archivista releva al programador de las tareas de oficina, sistematiza y garantiza un desempeño adecuado de esas tareas rutinarias que a menudo se descuidan, y mejora el activo más valioso del equipo – el producto del trabajo. Evidentemente, el concepto expuesto con anterioridad asume ejecuciones por lotes. Cuando se usan terminales interactivas, en particular aquellas sin salidas impresas, las funciones del archivista no disminuyen, pero cambian. Ahora registra todas las actualizaciones de las copias del programa del equipo a partir de copias de trabajo privadas, todavía se encarga de las ejecuciones por lotes, y usa sus propios medios interactivos para controlar la integridad y disponibilidad del producto en desarrollo.

El especialista en herramientas. Hoy es muy fácil contar con servicios de edición de archivos, edición de texto y depuración interactiva, de tal manera que un equipo rara vez necesitará su propia computadora y personal de operadores. Pero estos servicios deben estar disponibles con una respuesta incuestionablemente satisfactoria y confiable; y el cirujano debe ser el único juez de la idoneidad del servicio puesto a su disposición. Aparte, necesitará un especialista en herramientas, responsable de asegurar esta idoneidad del servicio básico y de la construcción, mantenimiento, y actualización de herramientas especiales – en su mayoría serán servicios de cómputo interactivo – necesarias para se equipo. Cada equipo necesitará su propio especialista en herramientas, independientemente de la calidad y confiabilidad de cualquier servicio proporcionado por la administración central, porque su trabajo es velar por las herramientas que *su* cirujano necesita o busca, sin tener en cuenta las necesidades de los demás equipos. El fabricante de herramientas a menudo construirá programas de soporte especializados, catálogo de procedimientos, y bibliotecas de macros.

El probador. El cirujano necesitará un adecuado banco de casos de prueba para probar piezas de su trabajo mientras los escribe, y después para la prueba general. El probador, es por la tanto, por un lado un adversario que idea casos de prueba del sistema a partir de las especificaciones funcionales, y por otro lado, un colaborador que idea datos de prueba para la depuración diaria. También podría planificar secuencias de pruebas y construir el andamiaje nece-

sario para las pruebas de componentes.

El abogado del lenguaje. Cuando apareció Algol, el personal empezó a reconocer que la mayoría de las instalaciones de computadoras tenían una o dos personas que se deleitaban en el dominio de las complejidades de los lenguajes de programación. Estos expertos resultaron ser muy útiles y ampliamente consultados. Este talento es bastante diferente del talento del cirujano, quien es principalmente un diseñador de sistemas y alguien que idea representaciones. El abogado del lenguaje puede encontrar una manera pulcra y eficiente de usar el lenguaje para realizar cosas difíciles, obscuras o truculentas. A menudo deberá hacer estudios breves (de dos o tres días) acerca de una buena técnica. Un abogado del lenguaje puede atender a dos o tres cirujanos.

Pues así es como 10 personas pueden contribuir en roles bien diferenciados y especializados en un equipo de programación construido bajo el modelo quirúrgico.

Cómo funciona

El equipo recién definido cumple con los propósitos de varias formas. Diez personas, siete de ellas profesionales, trabajan en un problema, pero el sistema es el producto de una sola mente – o a lo mucho dos, actuando *uno animo*.

En particular, observe las diferencias entre un equipo de dos programadores organizados convencionalmente y el equipo cirujano-copiloto. Primero, en el equipo convencional los participantes se dividen el trabajo, y cada uno es responsable del diseño e implementación de una parte del trabajo. En el equipo quirúrgico, el cirujano y el copiloto están cada uno al tanto de todo el diseño y todo el código. Esto ahorra el trabajo de la asignación de espacio de memoria, de los accesos a disco etc. Y también asegura la integridad conceptual del trabajo.

Segundo, en el equipo convencional los participantes son iguales, y las inevitables diferencias de opinión deben ser discutidas o negociadas. Puesto que el trabajo y los recursos están divididos, las diferencias de opinión están confinadas a la estrategia global y la interfaz, aunque estén compuestas por diferencias de interés – e.g., de quien cuyo espacio de memoria será usado para un búfer. En el equipo quirúrgico, no hay diferencias de interés, y las diferencias de opinión las resuelve el cirujano unilateralmente. Estas dos

diferencias – la falta de división del problema y la relación superior-subordinado hacen posible que el equipo quirúrgico actúe *uno animo*.

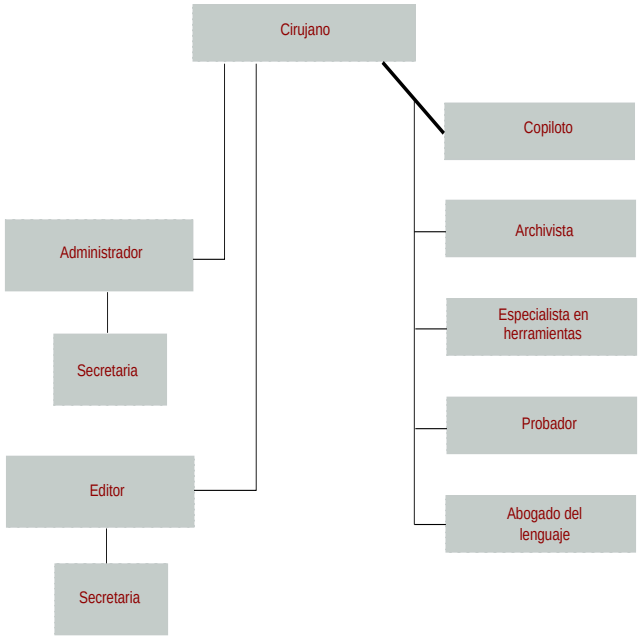


Fig. 3.1 Patrón de comunicación en equipos de programación de 10 personas

Incluso la especialización de funciones del resto del equipo es la clave para su eficiencia, y permite un patrón de comunicación radicalmente más simple entre sus miembros, como muestra la Fig. 3.1.

El artículo de Backer[3] reporta acerca de una sola prueba a pequeña escala del concepto de equipo. Para ese caso funcionó como se predijo, con resultados estupendamente buenos.

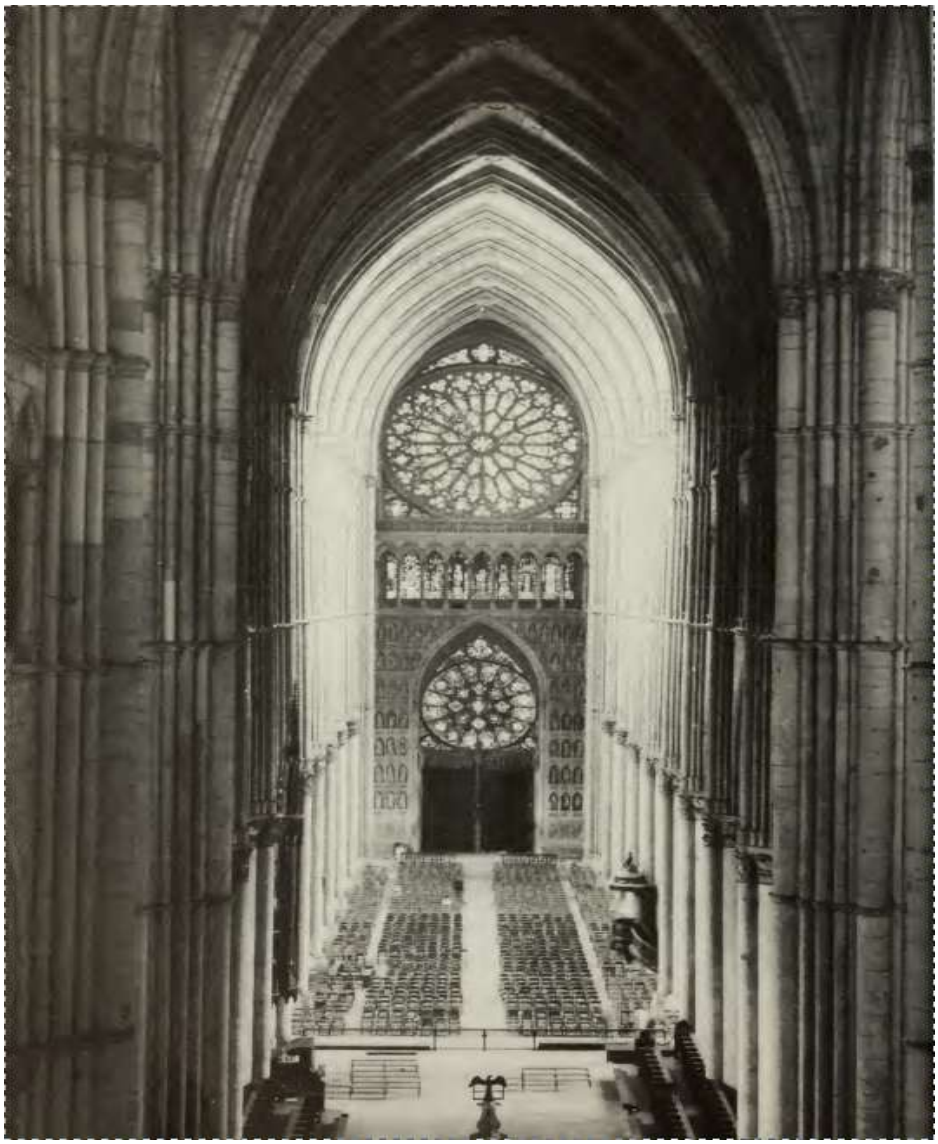
Ampliación

Al menos por el momento. Sin embargo, el problema es cómo construir cosas que tomen 5000 hombres-año, no cosas que tomen 20 o 30. Un equipo de 10 personas puede ser eficaz sin importar cómo se organice, siempre que *todo* el trabajo esté dentro de su ámbito. Pero ¿Cómo se usa el concepto de equipo quirúrgico en trabajos grandes cuando se involucran varios cientos de personas para llevar a cabo la tarea?

El éxito del proceso de ampliación depende del hecho de que la integridad conceptual de cada pieza haya sido mejorada radicalmente – ese número de mentes que determinan el diseño se ha divido entre siete. Así es posible poner 200 personas en un problema y encarar el problema de coordinar solo 20 mentes, las de los cirujanos.

Sin embargo, para el problema de coordinación se deben usar técnicas distintas, que se discuten en los siguientes capítulos. Aquí es suficiente decir que todo el sistema también debe tener integridad conceptual, y que se necesita un arquitecto del sistema que lo diseñe completamente, de forma descendente. Para hacer el trabajo manejable, es importante distinguir claramente entre arquitectura e implementación, el arquitecto del sistema debe restringirse él mismo escrupulosamente a la arquitectura. Sin embargo, tales roles y técnicas han demostrado ser viables y, asimismo, muy productivas.

4
Aristocracia, Democracia, y Diseño de Sistemas



4
Aristocracia, Democracia, y Diseño de Sistemas

Esta gran iglesia es una obra de arte incomparable. No existe aridez, ni confusión en los principios expuestos . . .
Es el cenit de un estilo, el trabajo de artistas que entendieron y asimilaron todos los éxitos de sus predecesores, aunque en completa posesión de las técnicas de su tiempo, las usaron sin la indiscreta exposición ni la injustificada proeza del talento.
Fue Jean d'Orbais quien sin duda concibió el plan general del edificio, un plan que ha sido respetado, al menos en sus elementos esenciales, por sus sucesores. Esta es una de las razones por la extrema coherencia y unidad del edificio

GUÍA DE LA CATEDRAL DE REIMS[1]

Fotografías de Emmanuel Boudot-Lamotte

Integridad Conceptual

La construcción de la mayoría de las catedrales Europeas muestran diferencias entre sus diversas partes, ya sea en el plan o en el estilo arquitectónico, pues fueron realizadas por diferentes constructores y en distintas épocas. Los constructores posteriores se vieron tentados a "mejorar" los diseños de los anteriores, para reflejar por un lado los cambios en la moda y por otro las diferencias en el gusto personal. Así, la tranquilidad del crucero Normando colinda y contradice la elevada nave Gótica, y el resultado proclama el orgullo de los constructores tanto como la gloria de dios.

En contra de estos, la unidad arquitectónica de Reims se yergue en un espléndido contraste. El placer que emociona a los espectadores proviene tanto de la integridad del diseño como de cualquiera de sus méritos particulares. Como menciona la guía turística, la integridad se llevó a cabo a través de la autoabnegación de ocho generaciones de constructores, cada una de las cuales sacrificó algo de sus propias ideas con tal de que el todo pudiera ser un diseño puro. El resultado proclama no sólo la gloria de dios, sino también su poder para salvar a los pecadores de su soberbia.

Aun cuando no dispongan de siglos para construir, la mayoría de los sistemas de programación reflejan una desunión conceptual mucho peor que las catedrales. Por lo general, esto no surge a partir de una sucesión en serie de diseñadores expertos, sino de la separación del diseño en muchas tareas llevadas a cabo por muchas personas.

Yo sostengo que la integridad conceptual es *la* consideración más importante en el diseño de sistemas. Es mejor tener un sistema que omita ciertos rasgos anómalos y mejoras, pero que refleje un conjunto de ideas de diseño, que tener uno que contenga muchas ideas buenas pero independientes y sin coordinación. En este capítulo y en los dos siguientes, examinaremos las consecuencias de este tema para el diseño de los sistemas de programación:

- ¿Cómo se logra la integridad conceptual?
- ¿No implica este argumento una élite, o una aristocracia de arquitectos, y una horda de implementadores plebeyos cuyos talentos creativos e ideas se han suprimido?

- ¿Cómo se evita que los arquitectos se sumerjan en lo desconocido con especificaciones no implementables o costosas?
- ¿Cómo se asegura que cada detalle insignificante de una especificación arquitectónica sea comunicado al implementador, sea entendido correctamente por él, e incorporado exactamente en el producto?

Alcanzando la Integridad Conceptual

El propósito de los sistemas de programación es hacer que la computadora sea fácil de usar. Para lograr esto, se suministran lenguajes y diversos medios que de hecho son programas invocados y controlados por las propias características del lenguaje. Pero estos medios se compran a cierto precio: la descripción externa de un sistema de programación es de diez a veinte veces mayor que la descripción externa del propio sistema de cómputo. El usuario descubre que es mucho más fácil especificar cualquier función excepcional, aunque hay mucho más de donde escoger, y muchas más opciones y formatos que recordar.

La facilidad de uso se mejora solo si el tiempo ganado en la especificación de la funcionalidad excede el tiempo invertido en aprender, recordar y en la búsqueda de manuales. Con los sistemas de programación modernos esta ganancia supera el costo, aunque en los años recientes la proporción entre ganancia y costo parece haber disminuido conforme se han ido agregado funcionalidades más complejas. Estoy cautivado por el recuerdo de la facilidad de uso de la IBM 650, incluso sin un programa ensamblador o ningún otro tipo de software.

Debido a que la facilidad de uso es el propósito, la proporción entre funcionalidad y complejidad conceptual es la prueba final del diseño de sistemas. Ni la funcionalidad, ni la simplicidad por sí solas definen un buen diseño.

Este punto se ha malentendido ampliamente. El Operative System/360 es aclamado por sus constructores como la máquina más fina jamás construida, debido a que indiscutiblemente tiene la mayor funcionalidad. Pues la medida de excelencia de sus diseñadores siempre fue la funcionalidad, no la simplicidad. Por otro lado, el Sistema de Tiempo-Compartido para la PDP-10 es aclamado por sus constructores como el más fino, debido a su simplicidad y la sobriedad de sus conceptos. Sin embargo, a través de cualquier medición su

funcionalidad no están ni siquiera en la misma clase que el OS/360. En cuanto tomamos la facilidad de uso como criterio, todos ellos parecen estar desbalanceados, y solo logran alcanzar la mitad de sus verdaderas metas.

Sin embargo, para un cierto nivel de funcionalidad, el mejor sistema es aquel en el cual se pueden especificar cosas con la mayor simplicidad y sencillez posibles. Pero la *simplicidad* no es suficiente. El lenguaje TRAC de Mooers y Algol 68 logran la simplicidad medida por el número de conceptos elementales distintos. Sin embargo, no son *sencillos*. Pues, para expresar cosas que con frecuencia uno quiere hacer se requiren combinaciones complicadas e imprevistas de los medios básicos. No es suficiente aprender los elementos y las reglas para combinarlas; también se debe aprender el uso idiomático, y toda una tradición de cómo se combinan los elementos en la práctica. La simplicidad y la sencillez provienen de la integridad conceptual. Cada parte debe reflejar las mismas filosofías y el mismo equilibrio en los propósitos. Incluso cada parte debe usar las mismas técnicas en la sintaxis y nociones análogas en la semántica. Por lo tanto, la facilidad de uso impone la unidad del diseño y la integridad conceptual.

Aristocracia y Democracia

La integridad conceptual a su vez obliga a que el diseño proceda de una sola mente, o de un muy pequeño número de mentes en sintonía.

Sin embargo, las presiones del calendario obligan a que la construcción del sistema requiera mucha manos. Se disponen de dos técnicas para resolver este dilema. La primera es una cuidadosas división del trabajo entre arquitectura e implementación. La segunda es la nueva forma de estructurar los equipos encargados de la implementación de la programación discutida en el capítulo anterior.

Una forma muy poderosa de lograr la integridad conceptual en proyectos muy grandes consiste en la separación del esfuerzo arquitectónico de la implementación. Yo mismo he visto su uso con gran éxito en la computadora Stretch de IBM y en la línea de productos de la computadora System/360. Y también su fracaso por la escasez de su puesta en práctica en el Operative System/360.

Por *arquitectura* de un sistema, quiero decir la especificación completa y detallada de la interfaz de usuario. Para una computadora este es el manual

de programación. Para un compilador es el manual del lenguaje. Para un programa de control son los manuales del lenguaje o los lenguajes utilizados para invocar sus funciones. Para el sistema completo es la unión de los manuales que el usuario debe consultar para realizar todo su trabajo.

El arquitecto de un sistema, como el arquitecto de un edificio, es el representante del usuario. Su trabajo es brindar un conocimiento profesional y técnico para apoyar el genuino interés del usuario, opuesto al interés del vendedor, fabricante, etc.[2]

Debemos distinguir cuidadosamente la arquitectura de la implementación. Como Blaauw afirma, "Donde la arquitectura dice *qué* sucede, la implementación dice *cómo* se hace para que suceda."[3] Da como un ejemplo sencillo el reloj, cuya arquitectura consta de la carátula, las manecillas y el botón de la cuerda. Cuando un niño aprende esta arquitectura, puede decir la hora tan fácilmente desde un reloj de pulsera como desde la torre de una iglesia. Sin embargo, la implementación y su realización, describen qué ocurre dentro de la caja – impulsada y controlada con exactitud por uno de tantos mecanismos.

Por ejemplo, en la System/360 una sola arquitectura de computadora se implementó de forma muy diferente en cada uno de los nueve modelos. Al contrario, se usó una sola implementación del flujo de datos, la memoria y el microcódigo de la Model 30, en distintos momentos para cuatro arquitecturas diferentes: una computadora System/360, un canal múltiple con hasta 224 subcanales lógicamente independientes, un canal selector y una computadora 1401.[4]

La misma distinción es igualmente aplicable a los sistemas de programación. Existe un estándar estadounidense del Fortran IV. Esta es la arquitectura para muchos compiladores. Muchas implementaciones son posibles dentro de esta arquitectura: texto en el núcleo o compilador en el núcleo, compilación rápida u optimización, dirigida por sintaxis o *ad-hoc*. Del mismo modo, cualquier lenguaje ensamblador o lenguaje de control de tareas admite muchas implementaciones de parte del ensamblador o del planificador.

Ahora podemos abordar ese asunto tan sensible de la aristocracia contra la democracia. ¿No son los arquitectos una nueva aristocracia, una élite intelectual, creada para decirles a esos pobres implementadores tontos qué hacer? ¿No ha sido secuestrado todo el trabajo creativo por esta élite, dejando a los implementadores como engranajes de una máquina? ¿No obtendríamos

un mejor producto recogiendo las buenas ideas de todo el equipo, siguiendo una filosofía democrática, en lugar de restringir el desarrollo de las especificaciones a unos cuantos?

Respecto a la última pregunta, es la más fácil. Ciertamente no voy a sostener que sólo los arquitectos tienen buenas ideas arquitectónicas. Con frecuencia, los conceptos nuevos vienen de un implementador o de un usuario. Sin embargo, toda mi propia experiencia me convence, y he tratado de demostrar, que la integridad conceptual de un sistema determina su facilidad de uso. Las buenas ideas y características que no se integran con los conceptos básicos del sistema es mejor omitirlas. Si aparecen muchas de tales ideas importantes pero incompatibles, se desecha todo el sistema y se empieza de nuevo en un sistema integrado con conceptos básicos diferentes.

Como cargo a la aristocracia, la respuesta debe ser sí y no. Sí, en el sentido de que deben ser pocos arquitectos, su producto debe perdurar más que el de un implementador, y el arquitecto se coloca en el foco de las fuerzas que, en última instancia, se deben resolver en el interés del usuario. Si un sistema implica tener integridad conceptual, alguien debe controlar los conceptos. Esa es una aristocracia que no necesita excusa.

No, porque el establecimiento de especificaciones externas no es un trabajo más creativo que el diseño de implementaciones. Es solo un trabajo creativo diferente. El diseño de una implementación, dada una arquitectura, requiere y permite tanto diseño creativo, como muchas ideas nuevas, y tanta destreza técnica como el diseño de las especificaciones externas. Asimismo, la relación costo-rendimiento del producto dependerá en gran medida del implementador, así como la facilidad de uso depende en gran medida del arquitecto.

Existen muchos ejemplos de otras artes y oficios que nos llevan a creer que la disciplina es buena para el arte. En efecto, el aforismo de un artista dice, "La forma es liberadora." Los peores edificios son aquellos cuyo presupuesto era demasiado grande para los propósitos que servían. La creatividad de Bach apenas parece haber sido acallada por la necesidad de producir una cantata de forma limitada cada semana. Estoy seguro de que la computadora Stretch habría tenido una mejor arquitectura si hubiera estado más restringida. Las restricciones impuestas por el presupuesto de la System/360 Model 30 fueron, en mi opinión, totalmente beneficiosas para la arquitectura de la Model 75.

Igualmente, he visto que el suministro externo de una arquitectura mejora, no limita, el estilo creativo del grupo de implementación. El grupo de inmediato se concentra en la parte del problema que nadie ha abordado, y los inventos empiezan a fluir. Dentro de un grupo de implementadores sin restricciones, la mayoría de las ideas y debates se centran en las decisiones arquitectónicas, y la implementación en sí recibe poca atención.[5]

Este efecto, que he visto muchas veces, fue confirmado por R. W. Conway, cuyo grupo en Cornell construyó el compilador PL/C para el lenguaje PL/I. Él afirma, "Finalmente, decidimos implementar el lenguaje sin cambios ni mejoras, porque los debates acerca del lenguaje hubieran agotado todo nuestro esfuerzo."[6]

¿Mientras Tanto Qué Hace el Implementador?

Cometer un error multimillonario es una experiencia muy humillante, aunque también es muy memorable. Recuerdo claramente la noche en que decidimos cómo organizar la escritura de las especificaciones externas para el OS/360. El gestor de arquitectos, el gestor de implementadores del programa de control y yo estuvimos machacando el plan, el calendario y la división de responsabilidades.

El gestor de arquitectos tenía 10 hombres capaces. Afirmó que ellos podían escribir las especificaciones y hacerlo correctamente. Esto tomaría diez meses, tres más de lo que permitía el calendario.

El gestor del programa de control tenía 150 hombres. Afirmó que ellos podían preparar las especificaciones, en coordinación con el equipo de arquitectos; estaría bien hecho y sería práctico, y podía terminarlo a tiempo. Además, si el equipo de arquitectos lo hacía, sus 150 hombres estarían sentados meneando los dedos por diez meses.

A esto, el gestor de arquitectos respondió que si yo le daba la responsabilidad al equipo del programa de control, el resultado de hecho *no* estaría a tiempo, sino que estaría también retrasado tres meses, y de mucha menor calidad. Lo hice, y así fue. Estaba en lo cierto en ambos aspectos. Más aún, la falta de integridad conceptual hizo que el sistema fuera mucho más costoso de construir y cambiar, y diría que añadió un año al tiempo asignado a la depuración.

Por supuesto que se involucraron muchos factores en esa decisión errada; pero el más abrumador fue el tiempo de calendario y lo atractivo de poner a esos 150 implementadores a trabajar. Ahora quiero hacer notar los riesgos fatales de este canto de sirenas.

De hecho, cuando se propuso que un equipo pequeño de arquitectos escribiera todas las especificaciones externas para una computadora o sistema de programación, los implementadores plantearon tres objeciones:

- Las especificaciones serían bastantes ricas en funcionalidad y no reflejarían las consideraciones prácticas del costo.
- Los arquitectos obtendrían toda la diversión creativa y excluirían la inventiva de los implementadores.
- Muchos implementadores tendrían que quedarse de brazos cruzados mientras las especificaciones pasan a través del estrecho embudo que es el equipo de arquitectos.

La primera de ellas es un peligro real, y será tratada en el próximo capítulo. Las otras dos son simples y puros espejismos. Como vimos anteriormente, la implementación es también una labor creativa de primer orden. La oportunidad de ser creativos e inventivos en la implementación no disminuye significativamente al trabajar dentro de unas especificaciones externas dadas, e incluso esa práctica puede mejorar la estructura de la creatividad. El producto completo sin duda lo hará.

La última objeción es de coordinación y sincronización. Una respuesta rápida es abstenerse de contratar implementadores hasta que las especificaciones estén completas. Esto se hace cuando se construye un edificio.

Sin embargo, en el negocio de los sistemas de computación el ritmo es más ágil, y uno desea comprimir el calendario lo más posible. ¿Cuánto pueden traslaparse las especificaciones y la construcción?

Como Blaauw señala, el esfuerzo creativo íntegro involucra tres fases distintas: arquitectura, implementación, y realización. De hecho, resulta que pueden empezar en paralelo y avanzar simultáneamente.

Por ejemplo, en el diseño de la computadora el implementador puede empezar tan pronto como tenga suposiciones relativamente vagas acerca del manual, ideas un tanto más claras acerca de la tecnología, y costos y objetivos de desempeño bien definidos. Puede comenzar a diseñar flujos de datos, secuencias de control, conceptos aproximados del embalaje, etc. Idea o adapta las herramientas necesarias, especialmente el sistema de registro, incluyendo el sistema de automatización del diseño.

Mientras tanto, con respecto a la realización, se debe diseñar, mejorar y documentar los circuitos, las tarjetas, los cables, los armazones, las fuentes de poder y las memorias. Este trabajo prosigue en paralelo con la arquitectura y la implementación.

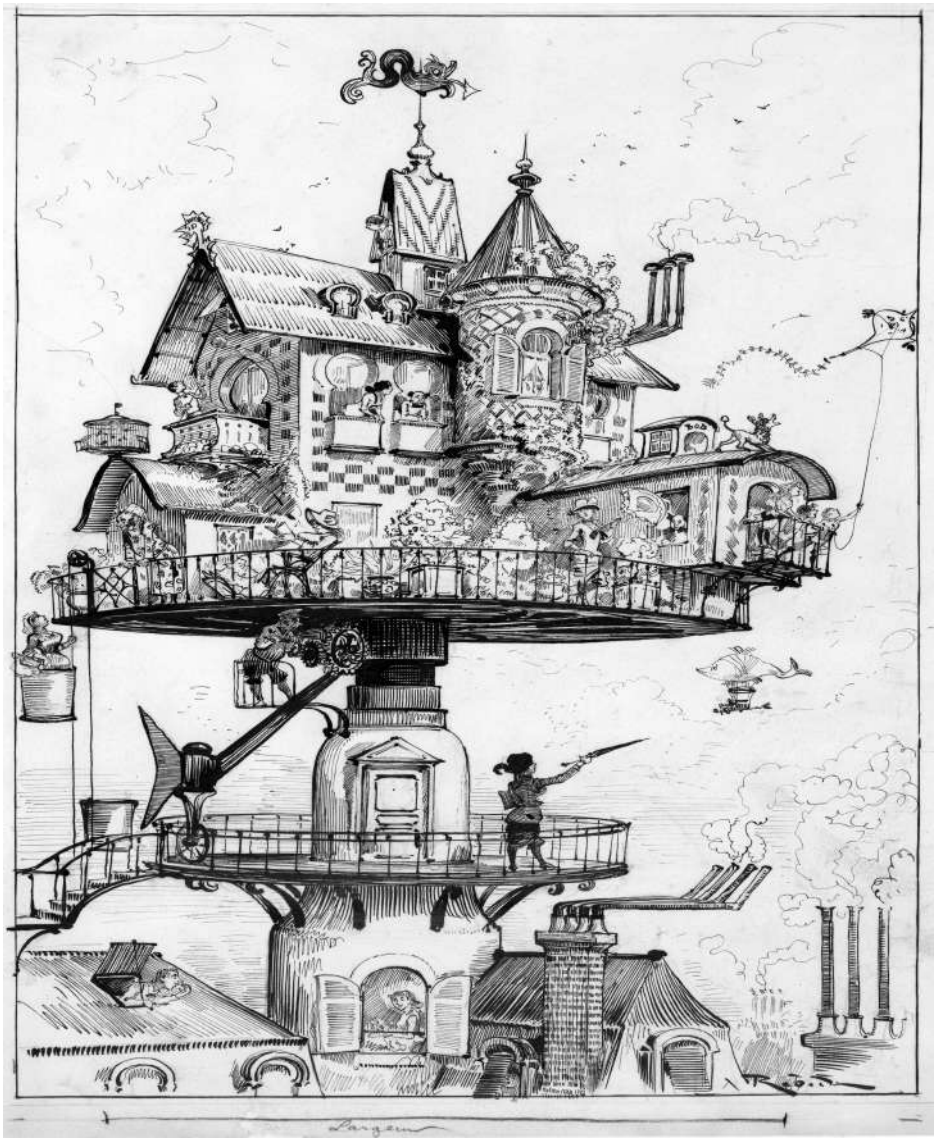
Lo mismo es cierto en el diseño del sistema de programación. Mucho antes de que se completen las especificaciones externas, el implementador tiene mucho que hacer. Puede proceder, dadas algunas vagas aproximaciones respecto a la funcionalidad del sistema que finalmente se incorporará a las especificaciones externas. Debe tener bien definidos los objetivos de espacio y tiempo. Debe conocer la configuración del sistema sobre el que se ejecutará su producto. Después puede empezar a diseñar los límites de los módulos, las estructuras de tablas, los desgloses de paso o fase, los algoritmos y todo tipo de herramientas. También, debe dedicar algo de tiempo a la comunicación con el arquitecto.

Mientras tanto, respecto a la realización también hay mucho por hacer. La programación también tiene una tecnología. Si la máquina es nueva, se debe trabajar mucho en las convenciones de las subrutinas, en las técnicas de supervisión y en los algoritmos de búsqueda y clasificación.[7]

La integridad conceptual requiere que un sistema refleje una sola filosofía y que la especificación vista por el usuario fluya de unas pocas mentes. Sin embargo, debido a la división real del trabajo entre arquitectura, implementación y realización, esto no implica que un sistema diseñado de esta forma demore más en construirse. La experiencia muestra lo contrario, que todo el sistema junto va más rápido y las pruebas demoran menos. De hecho, una

división del trabajo esparcida de forma horizontal ha sido reducida tajantemente a través de una división vertical del trabajo y el resultado son unas comunicaciones simplificadas de forma radical y una integridad conceptual mejorada.

5
El Efecto del Segundo-Sistema



5
El Efecto del Segundo-Sistema

Adde parvum parvo magnus acervus erit.
[Poco a poco se hace un gran montón.]

OVIDIO

Girando la casa por el tráfico. Litografía, Paris, 1882
De *Le Vingtième Siècle* por A. Robida

Si separamos la responsabilidad de la especificación funcional de la responsabilidad de construir un producto rápido y barato, ¿Cuál es el mecanismo que limita el entusiasmo inventivo del arquitecto?

La clave esencial es la exhaustiva, cuidadosa y comprensiva comunicación entre el arquitecto y el constructor. Sin embargo, existen cuestiones de grano más fino que merecen atención.

Procedimiento Interactivo para el Arquitecto

El arquitecto de un edificio trabaja en contra de un presupuesto, usa técnicas de estimación que luego son confirmadas o corregidas por las ofertas del contratista. Con frecuencia sucede que todas las propuestas exceden el presupuesto. Entonces el arquitecto mejora su estimación y reduce su diseño y realiza otra iteración. Quizá le pueda sugerir al contratista maneras de implementar su diseño de forma más barata de la que había ideado.

Un proceso análogo rige al arquitecto de un sistema de computación o de un sistema de programación. Tiene, además, la ventaja de obtener ofertas del contratista en muchas etapas tempranas del diseño, casi en cualquier momento que la solicite. Aunque, en general, tiene la desventaja de trabajar con un solo contratista, que aumenta o disminuye sus estimaciones para reflejar su acuerdo con el diseño. En la práctica, la comunicación pronta y continua puede brindar al arquitecto buenos indicios del costo y al constructor confianza en el diseño sin disipar la clara división de responsabilidades.

El arquitecto cuando se enfrenta con una estimación muy alta tiene dos opciones posibles: recortar el diseño o enfrentar la estimación sugiriendo implementaciones más baratas. La última es intrínsecamente una tarea emocionante. Ahora el arquitecto está retando al constructor acerca de la forma de hacer su trabajo de construcción. Para que esto tenga éxito, el arquitecto debe

- recordar que el constructor tiene la responsabilidad inventiva y creativa para realizar la implementación; así que el arquitecto sugiere, no dicta;
- estar siempre preparado para sugerir *una* manera de implementar cualquier cosa que se especifique y así también para aceptar cualquier otra

forma de cumplir con los objetivos;

- tratar de forma tranquila y privada tales sugerencias;
- estar listo a renunciar al crédito por las mejoras sugeridas.

El constructor, en general, será contrario a sugerencias de cambiar la arquitectura. A menudo estará en lo cierto – una característica menor puede tener grandes costos inesperados cuando se elabore la implementación.

Auto-Disciplina – El efecto del Segundo-Sistema

El primer trabajo de un arquitecto es adecuado para ser sobrio y limpio. Él sabe que no sabe qué es lo que está haciendo, así que lo hace con cuidado y con grandes restricciones.

A medida que el arquitecto diseña su primer trabajo, se le ocurre adorno tras adorno y embellecimiento tras embellecimiento. Todos estos quedan almacenados para usarlos la "próxima vez". Tarde o temprano el primer sistema está terminado, y el arquitecto, con firmeza, seguridad y una experiencia demostrada en esa clase de sistemas, está listo para construir un segundo sistema.

Este segundo es el sistema más peligroso que una persona jamás diseña. Cuando haga su tercer y los demás, sus experiencias previas se confirmarán mutuamente como las características generales de dichos sistemas, y sus diferencias identificarán esas partes de su experiencia que son particulares y no generalizables.

La tendencia general es sobrediseñar el segundo sistema, utilizando todas las ideas y adornos que cuidadosamente se hicieron a un lado en el primer sistema. El resultado, como dice Ovidio, es un "gran montón". Por ejemplo, considere la arquitectura de la IBM 709, más tarde incorporada en la 7090. Es una actualización, un segundo sistema de la muy exitosa y limpia 704. El conjunto de operaciones es tan rico y profuso que solo cerca de la mitad se usaba regularmente.

Consideremos como caso más sólido la arquitectura, la implementación, e incluso la realización de la computadora Stretch, un desahogo para los de-

seos inventivos reprimidos de mucha gente, y un segundo sistema para la mayoría de ellos. Como afirma Strachey en su reseña:

> *Tengo la impresión de que la Stretch es de alguna manera el fin de una tendencia de desarrollo. Al igual que algunos primeros programas de computación es inmensamente ingeniosa, inmensamente complicada y extremadamente eficaz, pero de alguna manera al mismo tiempo cruda, excesiva y torpe, y uno siente que debe haber una mejor forma de hacer las cosas.*[1]

El Operative System/360 fue el segundo sistema para la mayoría de sus diseñadores. Los grupos de diseñadores venían de construir el sistema operativo de disco 7010, el sistema operativo Stretch, el sistema de tiempo real Project Mercury y el IBSYS para la 7090. Casi nadie tenía experiencia con *dos* sistemas operativos previos.[2] Así que el OS/360 es un excelente ejemplo del efecto del segundo-sistema, un Stretch del arte del software al cual se aplican sin cambios tanto el elogio como el reproche de Strachey.

Por ejemplo, el OS/360 dedicaba 26 bytes a una rutina residente de forma permanente de cambio de fecha para el apropiado manejo del 31 de Diciembre en años bisiestos (cuando es el Día 366). Esto pudo haberse dejado al operador.

El efecto del segundo-sistema tiene otra manifestación algo distinta del puro adorno funcional. Y es una tendencia a refinar técnicas cuya misma existencia se ha vuelto obsoleta por los cambios en los supuestos básicos del sistema. El OS/360 tiene muchos ejemplos de este tipo.

Considere el editor de enlace, diseñado para cargar programas compilados de forma separada y resolver sus referencias-cruzadas. Más allá de esta función básica también maneja las superposiciones del programa. Es uno de los medios de superposición más finos jamás construidos. Permite estructurar la superposición externamente, en tiempo de enlace, sin necesidad de diseñarlo en el código fuente. Permite cambiar la estructura de la superposición de ejecución a ejecución sin recompilación. Provee una rica variedad de opciones y medios útiles. En cierto sentido es la culminación de años de desarrollo de la técnica de superposición estática.

Sin embargo, es también el último y más fino de los dinosaurios, porque pertenece a un sistema en el que la multiprogramación es el modo normal y la asignación dinámica del núcleo la premisa principal. Esto entra en conflicto directo con la noción de usar superposiciones estáticas. Cuánto mejor funcionaría el sistema si el esfuerzo dedicado al manejo de superposiciones se hubiera invertido en lograr que la asignación dinámica del núcleo y los medios de referencia-cruzada dinámica fueran realmente rápidos!

Además, el editor de enlace requiere tanto espacio que contiene en sí mismo muchas superposiciones que aun cuando se usa solo para enlazar sin el manejador de superposiciones, es más lento que la mayoría de los compiladores del sistema. La ironía de esto es que el propósito del enlazador es evitar la recompilación. Esto es como el patinador cuyo estómago se adelante a sus pies, la mejora del sistema avanzó hasta que las suposiciones acerca de éste fueron totalmente rebasadas.

El medio de depuración del TESTRAN es otro ejemplo de esta tendencia. Es la culminación de los medios de depuración por lotes, suministra una copia del estado del sistema realmente elegante y capacidades para el volcado de memoria. Usa el concepto de sección de control y una ingeniosa técnica generadora que permite el trazado selectivo y copia del estado del sistema sin sobrecarga del intérprete o recompilación. Los conceptos creativos del Share Operating System[3] para la 709 alcanzaron su máximo esplendor.

Mientras tanto, toda la idea de la depuración por lotes sin recompilación se estaba volviendo obsoleta. Los sistemas de computación interactivos, que usan intérpretes o compiladores incrementales, han resultado ser el desafío más importante. Pero incluso en los sistemas por lotes, la aparición de compiladores de rápida-compilación/lenta-ejecución ha hecho de la depuración a nivel código fuente y de la copia del estado del sistema la técnica preferida. Cuánto mejor hubiera resultado el sistema si se hubiera dedicado, cuanto antes y mejor, mayor esfuerzo al TESTRAN, que a la construcción de medios interactivos y de compilación-rápida!

Otro ejemplo más es el planificador, el cual proporciona realmente excelentes medios para la administración de un flujo de tareas por lotes fijo. En sentido estricto, este planificador es el refinado, mejorado y embellecido segundo sistema sucesor del 1410-1710 Disk Operative System, un sistema por lotes sin multiprogramación excepto para entradas y salidas y dirigido prin-

cipalmente a aplicaciones comerciales. Como tal, el planificador del OS/360 funciona bien. Aunque casi está totalmente influenciado por las necesidades del OS/360 de entrada de tareas remotas, multiprogramación, y subsistemas interactivos residentes de forma permanente. Y en efecto, el diseño del planificador hace estas cosas difíciles.

¿Cómo evita el arquitecto el efecto del segundo-sistema? Bien, obviamente no puede evitar su segundo sistema. Pero puede estar consciente de los peligros peculiares de dicho sistema, y ejercer una autodisciplina extra para evitar la ornamentación funcional y la extrapolación de funciones que son innecesarias por cambios en las suposiciones y propósitos.

Un hábito que abrirá los ojos del arquitecto es asignar a cada pequeña función un valor: la habilidad *x* no vale más que *m* bytes de memoria y *n* microsegundos por invocación. Estos valores guiarán las decisiones iniciales y servirán durante la implementación como una guía y alarma para todo.

¿Cómo evita el gestor del proyecto el efecto del segundo-sistema? Exigiendo un arquitecto experimentado que tenga al menos dos sistemas en su haber. Además, permaneciendo atento a las tentaciones particulares, puede hacer las preguntas correctas para así garantizar que los conceptos filosóficos y los objetivos se reflejen totalmente en los detalles del diseño.

6
Pasar la Voz

6
Pasar la Voz

Se sentará aquí y dirá, "Haz esto! Haz aquello!" Y
nada sucederá.

HARRY S. TRUMAN, ACERCA DEL PODER PRESIDENCIAL [1]

"Las Siete Trompetas" del *The Wells Apocalypse*, siglo 14
El Archivo Bettman

Supongamos que el gestor tiene arquitectos disciplinados y experimentados y muchos implementadores, ¿cómo garantiza el gestor que todos escuchen, entiendan e implementen las decisiones tomadas por los arquitectos? ¿Cómo puede un grupo de 10 arquitectos mantener la integridad conceptual de un sistema que está siendo construido por 1000 personas? Con este fin se desarrolló toda una tecnología para la labor de diseño del hardware de la System/360, y es igualmente aplicable a proyectos de software.

Especificaciones Escritas – el Manual

El manual, o la especificación escrita, es una herramienta necesaria, aunque no suficiente. El manual es la especificación *externa* del producto. Describe y prescribe cada detalle de lo que el usuario ve. Como tal, es el principal producto del arquitecto.

Su ciclo de preparación es iterativo, conforme la retroalimentación de los usuarios e implementadores muestre dónde el diseño es difícil de usar o construir. Por el bien de los implementadores es importante que se cuantifiquen los cambios – que en el calendario aparezcan versiones fechadas.

El manual no solo debe describir todo lo que el usuario ve, incluyendo todas las interfaces; también debe abstenerse de describir lo que no ve. Esa es la responsabilidad del implementador, y ahí no se debe restringir su libertad de diseño. El arquitecto debe estar siempre preparado para mostrar *una* implementación de cualquier característica que describa, aunque no debe intentar dictar *la* implementación.

El estilo debe ser preciso, completo y detallado con exactitud. Un usuario generalmente se referirá a una sola definición, así que es necesario repetir completamente todo lo indispensable y aún así todo debe ser coherente. Esto ocasiona que los manuales sean de lectura aburrida, pero la precisión es más importante que la vitalidad.

La coherencia del manual System/360's *Principles of Operation*, surge del hecho de que solo fue escrito por dos plumas; la de Gerry Blaauw y la de Andris Padegs. Aunque las ideas son de alrededor de 10 personas, el plasmar esas decisiones en especificaciones en prosa debe ser hecha sólo por una o dos personas, para mantener la coherencia de la prosa y del producto. Para la escritura de una definición se necesitará un montón de minidecisiones que no tienen la importancia de una discusión amplia. Por ejemplo, en la System/360

está el detalle de cómo poner el Código de Condición después de cada operación. *No* es trivial, sin embargo, es el principio de que tales mini-decisiones deben ser coherentes de principio a fin.

Creo que la pieza más fina de escritura de un manual que he visto es el Apéndice de Blaauw del *System/360 Principles of Operation*. Describe con cuidado y precisión los *límites* de la compatibilidad de la System/360. Define la compatibilidad, prescribe lo que se consigue, y enumera esas áreas de aspecto externo donde la arquitectura se enmudece intencionalmente y donde los resultados de un modelo pueden diferir de otros, donde una copia de un modelo dado puede diferir de otra copia, o donde una copia puede diferir incluso de sí misma después de un cambio de ingeniería. Este es el nivel de precisión al cual aspiran los escritores de manuales, y deben definir lo que *no* está prescrito tan cuidadosamente como lo que sí está.

Definiciones Formales

El Inglés o cualquier otro idioma, por su naturaleza, no es un instrumento de precisión para las definiciones formales. Por lo tanto, el escritor del manual debe forzarse él mismo y forzar su lenguaje para lograr la precisión requerida. Una alternativa interesante es usar una notación formal para tales definiciones. Después de todo, la precisión es el principal recurso, la *raison d'être* de las notaciones formales.

Examinemos las cualidades y debilidades de las definiciones formales. Como hemos indicado, las definiciones formales son precisas. Tienden a ser completas; los vacíos se hacen más visibles, y así se llenan más pronto. De lo que carecen es de inteligibilidad. Con la prosa en Inglés se pueden mostrar principios estructurales, delinear la estructura en etapas o niveles, y dar ejemplos. Se pueden marcar rápidamente excepciones y destacar contrastes. Y lo más importante, se puede explicar *por qué*. Las definiciones formales presentadas hasta ahora han inspirado maravillas por su elegancia y confianza en su precisión. Pero han exigido explicaciones en prosa para que su contenido sea más fácil de aprender y enseñar. Por estas razones, creo que en el futuro las especificaciones consistirán de una definición formal *y* una definición en prosa.

Un adagio antiguo advierte, "Nunca vayas al mar con dos cronómetros; lleva uno o tres." Lo mismo se aplica claramente a la prosa y a las definiciones

formales. Si tenemos ambas, el estándar debe ser una, y la otra la descripción derivada, claramente etiquetada como tal. Cualquiera de las dos puede ser el estándar principal. Algol 68 tiene una definición formal como estándar y una definición en prosa como descriptiva. PL/I tiene la prosa como estándar y la descripción formal como derivada. La System/360 tiene también una descripción en prosa como estándar con una descripción formal como derivada.

Disponemos de muchas herramientas para las definiciones formales. La notación de Backus-Naur es común para la definición del lenguaje, y es tratada ampliamente en la literatura.[2] La descripción formal del lenguaje PL/I utiliza nuevas ideas de sintaxis abstracta, y está descrita adecuadamente.[3] El lenguaje APL de Iverson ha sido usado para describir máquinas, en especial la IBM 7090[4] y la System/360.[5]

Bell y Newell han propuesto nuevas notaciones tanto para describir configuraciones como arquitecturas de máquinas, y las han ilustrado con varias computadoras, entre ellas la DEC PDP-8[6], la 7090[6] y la System/360.[7]

Casi todas las definiciones formales resultan para expresar o describir una implementación de un sistema de hardware o software cuyas especificaciones externas están prescribiendo. La sintaxis se puede describir sin esto, pero la semántica generalmente se define proporcionando un programa dado que realice la operación definida. Esto es, por supuesto, una implementación, y como tal prescribe en exceso la arquitectura. Así pues, se debe tener cuidado al señalar que la definición formal se aplica solo a las especificaciones externas, y se debe decir cuáles son éstas.

Una implementación no sólo es una definición formal, una implementación también se puede usar como una definición formal. Esta fue precisamente la técnica usada cuando se construyeron las primeras computadoras compatibles. La nueva máquina tenía que ser compatible con una máquina existente. ¿Era vago el manual en algunos puntos? "Pregúntele a la máquina!" Se crearía un programa de prueba para determinar el comportamiento, y la nueva máquina se construiría para hacerla compatible.

Un simulador de un sistema de hardware o software puede servir exactamente en la misma forma. Es una implementación; funciona. Así que todas las preguntas de la definición se pueden resolver probándolas.

Usar una implementación como una definición tiene algunas ventajas. Se pueden aclarar todas las preguntas sin ambigüedades a través de experi-

mentos. El debate nunca es necesario, por lo que las respuestas son rápidas. Las respuestas son siempre tan precisas como se quiera, y siempre son correctas, por definición. En oposición a estos se tiene un tremendo conjunto de desventajas. La implementación puede prescribir en exceso incluso las especificaciones externas. La sintaxis inválida siempre produce un resultado; en un sistema controlado ese resultado es una indicación de invalidez *y nada más*. En un sistema no controlado pueden aparecer todo tipo de efectos secundarios, y estos pudieron haber sido usados por los programadores. Por ejemplo, cuando emprendimos la simulación de la IBM 1401 sobre la System/360, resultó que habían 30 "curiosidades" – efectos secundarios de supuestas operaciones inválidas – que se habían usado ampliamente y tuvieron que considerarse como parte de la definición. La implementación como una definición sobre prescrita; no sólo diría lo que la máquina debe hacer, sino también diría mucho acerca de cómo se tenía que hacer.

Así también, la implementación a veces dará respuestas inesperadas y no planeadas cuando se hagan preguntas ingeniosas, y resultará que la definición *de facto* generalmente será poco elegante en estas peculiaridades precisamente porque nunca recibieron atención alguna. Esta falta de elegancia a menudo resultará ser lenta o costosa de duplicar en otra implementación. Por ejemplo, algunas computadoras dejan basura en el registro del multiplicando después de una multiplicación. La naturaleza precisa de esta basura resulta ser parte de la definición *de facto*, además duplicándola puede impedir el uso de un algoritmo de multiplicación más veloz.

Finalmente, el uso de una implementación como definición formal es particularmente susceptible a confusión con respecto a si la descripción en prosa o la descripción formal es de hecho el estándar. Esto es especialmente cierto en las simulaciones. También hay que abstenerse de realizar modificaciones a la implementación mientras sirve como estándar.

Incorporación Directa

El arquitecto del sistema de software dispone de una técnica encantadora para difundir y ejecutar definiciones. Es especialmente útil para establecer la sintaxis, si no es que la semántica, de las interfaces entre módulos. Esta técnica es para diseñar la declaración del pase de parámetros o almacenamiento compartido, y requiere que las implementaciones incluyan esa declaración a

través de una operación en tiempo de compilación (un macro o un `%INCLUDE` en PL/I). Si, además, toda la interfaz está referenciada solo a través de nombres simbólicos, se puede modificar la declaración añadiendo o insertando nuevas variables únicamente recompilando, sin alterar el programa en uso.

Reuniones de Trabajo y Tribunales

No hace falta decir que las reuniones son necesarias. Las cientos de consultas de persona a persona, deben complementarse con reuniones más largas y formales. Descubrimos que dos niveles de reuniones son útiles. La primera es una reunión de trabajo semanal de medio día de todos los arquitectos, más los representantes oficiales de los implementadores de hardware y software, y los planificadores de mercado. Lo preside el jefe de arquitectos del sistema.

Todos pueden proponer problemas o cambios, pero las propuestas normalmente se distribuyen por escrito antes de las reuniones. Un nuevo problema generalmente se discute un rato. Se hace énfasis en la creatividad, en lugar de las decisiones únicamente. El grupo intenta idear muchas soluciones a los problemas, luego se pasan unas cuantas soluciones a uno o más arquitectos para su detalle en un manual de propuestas de cambio redactado meticulosamente.

Luego, se presentan propuestas de cambio minuciosas para la toma de decisiones. Se difunden y son revisadas cuidadosamente por implementadores y usuarios y se esbozan con claridad los pros y los contras. Si surge un consenso, tanto mejor. Si no, el jefe de arquitectos decide. Se conservan actas y las decisiones se difunden de manera formal, puntual y amplia.

Las decisiones que resultan de las reuniones semanales dan resultados rápidos y permiten que el trabajo avance. Si alguien no está *muy* satisfecho, puede apelar inmediatamente ante el gestor de proyecto, aunque esto es muy raro que suceda.

Los frutos de estas reuniones brotan de varias fuentes:

1. El mismo grupo – de arquitectos, usuarios e implementadores – se reúne semanalmente durante meses. No se requiere tiempo para actualizarlos.

2. El grupo es brillante, capaz, bien versado en estas cuestiones e involucrado profundamente en el resultado. Nadie tiene un papel de "asesor". Todos están autorizados para hacer acuerdos vinculantes.

3. Cuando se plantean problemas, se buscan soluciones tanto dentro como fuera de las fronteras obvias.

4. La formalidad de las propuestas escritas centra la atención, obliga a la toma de decisiones, y evita inconsistencias en la redacción del borrador del comité.

5. La evidente atribución de poder de decisión al jefe de arquitectos evita concesiones y demoras.

Conforme pasa el tiempo, algunas decisiones no se asumen. Algunos asuntos menores no han sido aceptados con entusiasmo por uno u otro participante. Otras decisiones han desarrollado problemas imprevistos, y a veces las reuniones semanales no estuvieron de acuerdo en reconsiderarlas. De esta forma, acumulamos una lista de reclamos menores, cuestiones no resueltas o desaveniencias. Para resolver esto, celebramos sesiones anuales del tribunal supremo, que suelen durar dos semanas. (Me gustaría mantenerlos cada seis meses si lo hiciera nuevamente.)

Estas sesiones se celebraban justo antes de las principales fechas de congelación del manual. Entre los presentes se encontraban no solo los arquitectos y los representantes de los programadores e implementadores de la arquitectura, sino también los gestores de programación, los de mercadotecnia y los implementadores. El gestor de proyecto de la Sytem/360 presidía. La agenda constaba normalmente de alrededor de 200 temas, en su mayoría de menor importancia, que se enlistaban en diagramas pegados en carteles alrededor del salón. Se escuchaban a todos los bandos y se decidía. Por el milagro de la edición de texto computarizada (y mucho por el excelente trabajo del personal), todos los participantes encontraban un manual actualizado cada mañana en su asiento, que plasmaba las decisiones tomadas el día anterior.

Estos "festivales de Otoño" fueron útiles no solo para la toma de decisiones, sino también para que fueran aceptadas. Todos eran escuchados, todos participaban, todos entendían mejor las intrincadas restricciones e interrelaciones entre las decisiones.

Implementaciones Múltiples

Los arquitectos de la Sytem/360 tenían dos ventajas casi sin precendentes: suficiente tiempo para trabajar cuidadosamente, e igual influencia política que los implementadores. El suministro de tiempo suficiente provino del calendario de la nueva tecnología; la igualdad política provino de la construcción simultánea de implementaciones múltiples. La necesidad de una compatibilidad estricta entre ellas sirvió como el mejor agente impositivo posible para las especificaciones.

En la mayoría de los proyectos llega el día en que se descubre que la máquina y el manual no están de acuerdo. Cuando la confrontación continúa, generalmente el manual pierde, ya que se puede cambiar de forma más rápida y barata que la máquina. Sin embargo, no tanto cuando existen múltiples implementaciones. Entonces las demoras y los costos asociados con la reparación de la máquina descarriada pueden ser superados por las demoras y costos en la revisión de las máquinas que siguieron fielmente el manual.

Podemos aplicar esta misma idea de forma fructífera siempre que estemos definiendo un lenguaje de programación. Uno puede estar seguro que tarde o temprano se tendrán que construir varios intérpretes o compiladores para cumplir diversos objetivos. La definición será más limpia y la disciplina más estricta si desde un inicio se construyen dos implementaciones al menos.

La Bitácora Telefónica

A medida que avanza la implementación, surgen incontables preguntas de interpretación arquitectónica, sin importar cuán precisa sea la especificación. Sin duda, muchas de dichas preguntas necesitan ser elaboradas y aclaradas en el texto. Otras únicamente reflejan malentendidos.

Sin embargo, es importante animar al implementador confundido a telefonear al arquitecto responsable y preguntarle, en lugar de adivinar y continuar. Es igualmente vital reconocer que las respuestas a dichas preguntas son declaraciones arquitectónicas *ex cathedra* que deben comunicarse a todos.

Un mecanismo útil es una *bitácora telefónica* que sea mantenida por el arquitecto. En ella registra cada pregunta y cada respuesta. Cada semana las bitácoras de varios arquitectos se concatenan, se reproducen y se distribuyen

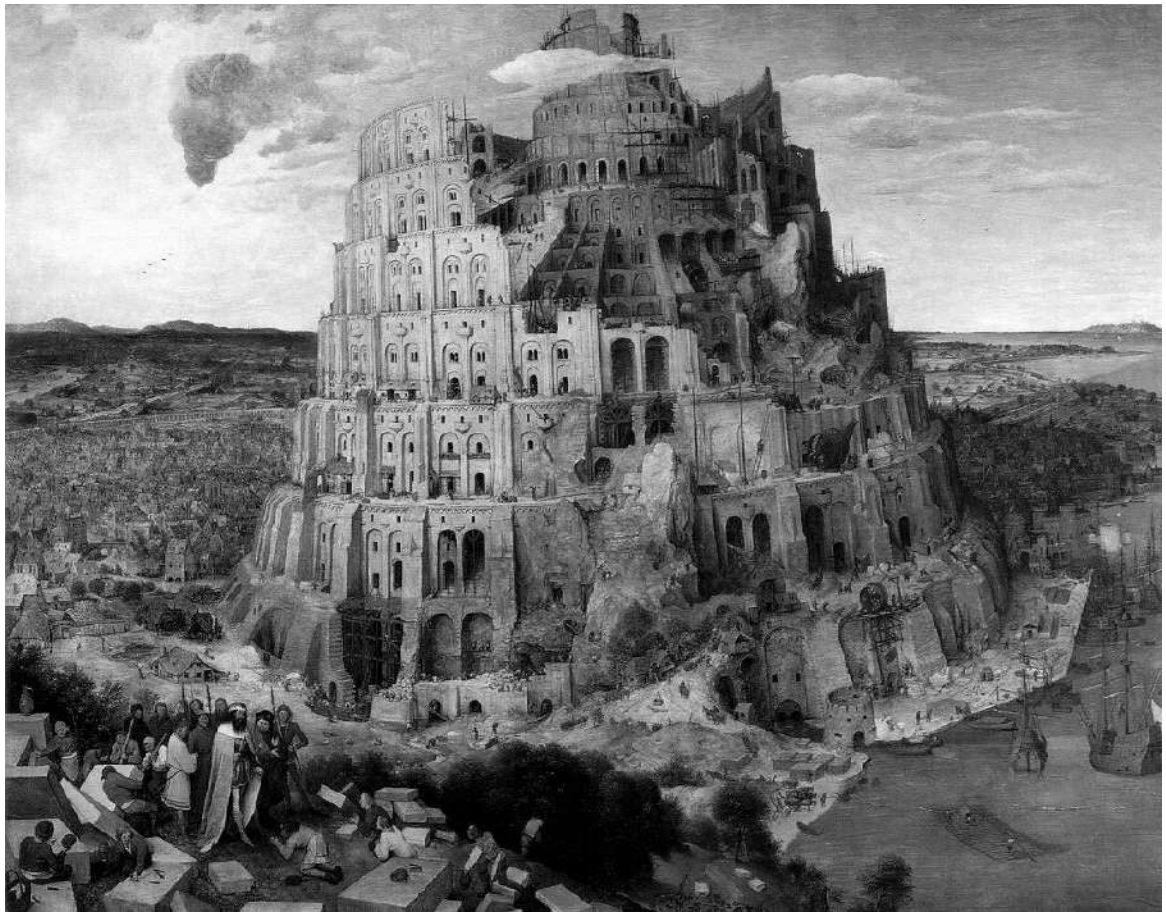
entre usuarios e implementadores. Si bien este mecanismo es bastante informal, es rápido y completo.

Prueba del Producto

El mejor aliado de un gestor de proyecto es su adversario cotidiano, y es la organización de un probador del producto independiente. Este grupo verifica las computadoras y los programas en contra de las especificaciones y sirve como abogado del diablo, señala cada defecto y discrepancia concebible. Toda organización de desarrollo necesita tal grupo de auditoría técnica independiente para mantener su credibilidad.

En última instancia, el cliente es el auditor independiente. A la luz del implacable uso en el mundo real, cada defecto se hará visible. El grupo de prueba del producto es por tanto el sucedáneo del cliente, está especializado en encontrar defectos. Una y otra vez, el prolijo probador del producto encontrará lugares donde no se pasó la voz, donde las decisiones de diseño no fueron entendidas apropiadamente o implementadas exactamente. Por este motivo, dicho grupo de prueba es un eslabón necesario en la cadena a través de la cual se pase la voz de diseño, un eslabón que debe operar pronta y simultáneamente con el diseño.

7
¿Por Qué Fracasó la Torre de Babel?



7
¿Por Qué Fracasó la la Torre de Babel?

Ahora bien, toda la tierra continuaba siendo de un solo lenguaje y de un solo conjunto de palabras. Y aconteció que, al ir viajando hacia el este, finalmente descubrieron una llanura-valle en la tierra de Sinar, y se pusieron a morar allí. Y empezaron a decirse, cada uno al otro: ¡Vamos! Hagamos ladrillos y cozámoslos con un procedimiento de quema. De modo que el ladrillo les sirvió de piedra, pero el betún les sirvió de argamasa. Entonces dijeron: ¡Vamos! Edifiquémonos una ciudad y también una torre con su cúspide en los cielos, y hagámonos un nombre célebre, por temor de que seamos esparcidos por toda la superficie de la tierra. Y el señor procedió a bajar para ver la ciudad y la torre que los hijos de los hombres habían edificado. A continuación dijo el señor: ¡Mira! Son un solo pueblo y hay un solo lenguaje para todos ellos, y esto es lo que comienzan a hacer. Pues, ahora no hay nada que tengan pensado hacer que no les sea posible lograr. ¡Vamos! Bajemos y confundamos allí su lenguaje para que no escuche el uno el lenguaje del otro.Por consiguiente, el señor los esparció desde allí sobre toda la superficie de la tierra, y poco a poco dejaron de edificar la ciudad.

GENESIS 11:1-8

"Turmbau zu Babel," P. Breughel, el Viejo, 1563
Museo Kunsthistorisches, Viena

Una Auditoría de Gestión del Proyecto Babel

De acuerdo al relato del Génesis, la torre de Babel fue el segundo gran emprendimiento de ingeniería del hombre, después del arca de Noé. Babel fue el primer fiasco de ingeniería.

La leyenda es profunda e instructiva en varios niveles. Sin embargo, examinémosla simplemente como un proyecto de ingeniería, y observemos qué lecciones de gestión podemos aprender. ¿Qué tan bien estaba equipado su proyecto con los prerrequisitos para el éxito? ¿Los tenían?

1. ¿Una *misión clara?* Sí, aunque ingenuamente imposible. El proyecto fracasó mucho antes de toparse con esta limitación fundamental.
2. *¿Mano de obra?* Abundante
3. *¿Materiales?* La arcilla y el asfalto son abundantes en Mesopotamia
4. *¿Tiempo* suficiente? Sí, no existe indicio de ninguna restricción de tiempo
5. *¿Tecnología* adecuada? Sí, la estructura piramidal o cónica es intrínsecamente estable y distribuye bien las carga de compresión. Claramente la albañilería estaba bien entendida. El proyecto fracasó antes de chocar con las limitaciones tecnológicas.

Ahora bien, si tenían todas estas cosas, ¿por qué fracasó el proyecto? ¿Dónde estuvieron sus carencias? En dos aspectos – *la comunicación*, y su consecuencia, *la organización*. Fueron incapaces de hablar entre ellos; por lo tanto no pudieron coordinarse. Cuando falló la coordinación, el trabajo se detuvo. Leyendo entre líneas deducimos que la falta de comunicación condujo a disputas, malos sentimientos y celos de grupo. Al poco tiempo los clanes empezaron a apartarse y prefirieron el aislamiento a la disputa.

La Comunicación en los Proyectos de Programación Grandes

Así es actualmente. El desastre del calendario, los desajustes funcionales y los errores del sistema todos surgen porque la mano izquierda no sabe lo que hace la derecha. Conforme avanza el trabajo, los distintos grupos cambian lentamente las funciones, los tamaños, las velocidades de sus propios programas, y cambian explícita o implícitamente sus supuestos acerca de las entradas disponibles y los usos que deben hacerse de las salidas.

Por ejemplo, el implementador de una función de superposición de programas puede toparse con problemas y reducir la *velocidad* basándose en estadísticas que muestran cuán rara vez aparecerá esta función en los programas de aplicación. Mientras tanto, de regreso a casa, sus vecinos podrían estar diseñando una parte importante del supervisor tal que dependa de manera crítica de la velocidad de esta función. En sí, este cambio en la velocidad viene a ser un gran cambio en la especificación, y debe ser comunicado ampliamente y ponderado desde el punto de vista del sistema.

¿Cómo, entonces, se debe comunicar un equipo con otro? De tantas formas como sea posible.

- *Informalmente.* Un buen servicio telefónico y una clara definición de las dependencias intergrupales fomentarán las cientos de llamadas de las que depende la interpretación común de los documentos escritos.
- *Reuniones.* Son invaluables las reuniones habituales del proyecto, con uno y otro equipo, que brinden información técnica. De esta forma se disipan cientos de malentendidos menores.
- *Libro de Trabajo.* Se debe empezar un libro de trabajo formal del proyecto desde un principio. Esto merece una sección aparte.

El Libro de Trabajo del Proyecto

Qué es. El libro de trabajo del proyecto no es tanto un documento separado sino una estructura impuesta a los documentos que el proyecto producirá de todos modos.

Todos los documentos del proyecto deben ser parte de esta estructura. Esto incluye objetivos, especificaciones externas, especificaciones de interfaces, estándares técnicos, especificaciones internas y memorandos administrativos.

Por qué. La prosa técnica es casi inmortal. Si se examina la genealogía de un manual de usuario de una pieza de hardware o software, se pueden rastrear no sólo las ideas, sino también muchas de las mismas oraciones y párrafos hasta los primeros memorandos proponiendo el producto o explicando el primer diseño. Para el escritor de documentos técnicos, el frasco de pegamento es tan poderoso como la pluma.

Ya que esto es así, y puesto que los manuales de calidad del producto de mañana crecerán a partir de las notas de hoy, es muy importante obtener la estructura correcta de la documentación. El diseño temprano del libro de trabajo del proyecto nos garantiza que la propia estructura de la documentación se hizo de forma organizada, no descuidada. Además, la creación de una estructura moldea la escritura posterior en segmentos que se ajustan a esa estructura.

El segundo motivo para el libro de trabajo es el control de la distribución de la información. El problema no es restringir la información, sino asegurar que la información relevante llegue a todos los que la necesiten.

El primer paso es numerar todos los memorandos, de tal manera que se dispongan de listas ordenadas de títulos y cada trabajador pueda ver si encuentra lo que busca. La organización del libro de trabajo va más allá de esto al establecer una estructura de árbol de memorandos. La estructura de árbol permite que las listas de distribución sean mantenidas por los subárboles, si así se desea.

Procedimiento. Como con muchos problemas de gestión de la programación, el problema del memorando técnico empeora de forma no lineal a medida que aumenta el tamaño. Con 10 personas, los documentos simplemente se pueden numerar. Con 100 personas, generalmente bastarán varias secuencias lineales. Con 1000 personas, se dispersan inevitablemente sobre varios espacios físicos, la *urgencia* por un libro de trabajo organizado aumenta, como también aumenta el *tamaño* del libro de trabajo. ¿Entonces, cómo se debe manejar este procedimiento?

Pienso que esto se hizo bien en el proyecto del OS/360. O. S. Locken insistió encarecidamente por un libro de trabajo bien organizado, él había visto su eficacia en su proyecto previo, el sistema operativo 1410-7010.

Rápidamente decidimos que *cada* programador debería ver *todo* el material, i.e., debería tener una copia del libro de trabajo en su propia oficina.

La actualización oportuna es de suma importancia. El libro de trabajo debe estar al corriente. Esto es muy difícil de hacer ya que debido a los cambios constantes todos los documentos se deben volver a escribir. Sin embargo, en un libro de hojas sueltas solo es necesario cambiar páginas. Disponíamos de un sistema de edición de texto por computadora, y demostró ser invaluable para el mantenimiento puntual. Las impresiones offset maestras se prepara-

ban directamente en la impresora de la computadora, y el tiempo de respuesta fue de menos de un día. Sin embargo, el destinatario de todas estas páginas actualizadas tiene un problema de asimilación. Cuando recibe por primera vez una página cambiada, quiere saber, "¿Qué se ha cambiado?" Cuando lo consulta más tarde, quiere saber, "¿Cuál es la definición el día de hoy?"

Esta última exigencia se cumple por el mantenimiento continuo del documento. Resaltar los cambios requiere otros pasos. Primero, se debe marcar en la página el texto modificado, e.g., mediante una barra vertical en el margen a lo largo de cada línea alterada. Segundo, se debe distribuir con las nuevas páginas una síntesis, un resumen de cambios escrito de forma separada que enumere los cambios y observaciones acerca de su importancia.

Nuestro proyecto no había andado ni seis meses antes de enfrentarnos a otro problema. El libro de trabajo era de un grosor de cinco pies! Si hubiéramos apilado las 100 copias de servicio de los programadores en nuestras oficina del edificio del Time-Life en Manhattan, se habrían elevado por encima del edificio mismo. Además, la distribución diaria de cambios promediaba 2 pulgadas, unas 150 páginas para ser intercaladas en el conjunto. El mantenimiento del libro de trabajo empezó a tomar un tiempo significativo en el trabajo diario.

En este punto cambiamos a microfichas, un cambio que ahorró un millón de dólares, incluso nos permitió costear un lector de microfichas para cada oficina. Fuimos capaces de organizar un excelente cambio en la producción de microfichas; el libro de trabajo se encogió de tres pies cúbicos a una sexta parte de un pie cúbico y, lo que es más significativo, las actualizaciones aparecieron en paquetes de cien páginas, reduciendo el problema del intercalado cien veces.

La microficha tiene sus inconvenientes. Desde el punto de vista del gestor el inconveniente de intercalar las páginas aseguraba que los cambios fueran *leídos*, que era el propósito del libro de trabajo. La microficha facilitaría bastante el mantenimiento del libro de trabajo, a condicion de que la ficha de actualización se distribuya con un documento en papel enumerando los cambios.

Además, una microficha no puede ser fácilmente resaltada, marcada y comentada por el lector. Los documentos con los cuales el lector ha interactuado son más eficaces para el autor y mas útiles para el lector.

En general, pienso que la microficha fue un mecanismo muy oportuno, y para proyectos muy grandes lo recomendaría en lugar de un libro de trabajo en papel.

¿Cómo se haría actualmente? Con la actual tecnología de sistemas disponible, creo que la técnica adecuada es el mantenimiento del libro de trabajo en un archivo de acceso directo, marcado con barras de cambio y fechas de revisión. Cada usuario lo consultaría desde una terminal de pantalla (las máquinas de escribir son muy lentas). Un resumen de cambios, preparado diariamente, se almacenaría en forma de pila (LIFO) en un punto de acceso fijo. Probablemente el programador lo leería diariamente, pero si falla un día solo deberá leer más tiempo al siguiente día. Mientras lee el resumen de cambios, puede interrumpir su lectura y consultar el propio texto modificado.

Note que el propio libro de trabajo no ha cambiado. Todavía es la colección de documentos de todo el proyecto, estructurado de acuerdo a un diseño prolijo. El único cambio está en los procedimientos de distribución y consulta. D. C. Engelbart y sus colegas en el Stanford Research Institute han construido dicho sistema y lo usan para construir y mantener la documentación de la red ARPA.

D.L. Parnas de la Universidad Carnegie-Mellon ha propuesto una solución aún más radical.[1] Su tesis es que el programador es más eficaz si se lo resguarda de, en lugar de exponerlo a los detalles de construcción de las piezas del sistema que no sean las suyas. Esto presupone que todas las interfaces están definidas de forma completa y precisa. A pesar de que eso es definitivamente un diseño sano, depender de su perfecta realización es una receta para el desastre. Un buen sistema de información por un lado expone los errores de las interfaces y por otro estimula su corrección.

La Organización en un Proyecto de Programación Grande

Si en un proyecto existen n trabajadores, existen $(n^2 - n)/2$ interfaces a través de las cuales puede haber comunicación, y hay potencialmente 2^n equipos dentro de los cuales debe existir coordinación. El propósito de la organización es reducir la cantidad de comunicación y coordinación necesaria; por lo tanto, la organización es un ataque radical a los problemas de comunicación tratados arriba.

Los medios por los cuales se evita la comunicación son *la división del trabajo* y *la especialización de funciones*. La estructura tipo árbol de las organizaciones refleja la necesidad cada vez menor de una comunicación detallada cuando se aplica la división y la especialización del trabajo.

De hecho, en realidad la organización tipo árbol surge como una estructura de autoridad y responsabilidad. El principio de que nadie puede servir a dos amos dicta que la estructura de autoridad sea tipo árbol. Aunque la estructura de comunicación no está tan restringida, y el árbol es una aproximación apenas aceptable de la estructura de comunicación, que es una red. Las deficiencias del enfoque tipo árbol dan origen a grupos de empleados, a grupos de trabajo, comités e incluso a la organización tipo matriz utilizada en muchos laboratorios de ingeniería.

Consideremos una organización de programación tipo árbol, y examinemos los ingredientes que cualquier subárbol debe tener para ser eficaz. Ellos son:

1. una misión
2. un productor
3. un director técnico o arquitecto
4. un calendario
5. una división del trabajo
6. definiciones de interfaz entre las partes

Todo esto es obvio y normal excepto la distinción entre el productor y el director técnico. Consideremos primero los dos roles, y luego su relación entre ellos.

¿Cuál es el papel del productor? Arma el equipo, divide el trabajo y establece el calendario. Consigue los recursos necesarios y continúa haciéndolo. Esto significa que una gran parte de su papel es la comunicación externa, hacia arriba y a los lados. Establece el patrón de comunicación e informes dentro

del equipo. Por último, garantiza que el calendario se cumpla, moviendo los recursos y la organización para responder al cambio de circunstancias.

¿Qué hay del director técnico? Es el que concibe el diseño que se construirá, identifica sus partes secundarias, especifica cómo se verá desde el exterior, y esboza su estructura interna. Proporciona unidad e integridad conceptual a todo el diseño; de esta manera, actúa como un límite sobre la complejidad del sistema. A medida que surgen problemas técnicos particulares, inventa soluciones o cambia el diseño del sistema cuando sea necesario. Es, en la encantadora frase de Al Capp, "nuestro agente infiltrado haciendo el trabajo sucio." Sus comunicaciones son principalmente dentro del equipo. Su trabajo es casi completamente técnico.

Ahora está claro que las cualidades requeridas para estos dos roles son bastante diferentes. Estas cualidades vienen en muchas combinaciones diferentes; esa combinación especial plasmada en el productor y el director debe regir la relación entre ellos. Las organizaciones se deben diseñar alrededor de las personas disponibles; nadie se ajusta a organizaciones puramente teóricas.

Se pueden dar tres tipos de relaciones, y las tres se hallan en prácticas exitosas.

El productor y el director técnico pueden ser la misma persona. Esto se puede hacer fácilmente en equipos muy pequeños, quizás de tres a seis programadores. Es muy poco viable en proyectos más grandes, por dos razones. Primero, rara vez se encuentra la persona con una sólida capacidad para la gestión y una sólida capacidad técnica. Los pensadores son raros; lo emprendedores son más raros; y los pensadores-emprendedores son los más raros.

Segundo, en un proyecto más grande cada uno de estos papeles es necesariamente más que un trabajo de tiempo completo. Es difícil para el productor delegar suficientes obligaciones para darse algo de tiempo técnico. Y para el director es imposible delegar las suyas sin comprometer la integridad conceptual del diseño.

El productor puede ser el jefe, el director su mano derecha. En este caso, la dificultad es establecer la *autoridad* del director para la toma de decisiones técnicas sin afectar su tiempo que lo pondría en la cadena de mando

de la gestión.

Obviamente el productor debe proclamar la autoridad técnica del director, y debe apoyarlo en una proporción extremadamente alta cuando se presenten los casos de prueba. Para que esto sea posible, el productor y el director deben tener una visión semejante sobre los principios técnicos fundamentales; deben discutir en privado sobre las principales cuestiones técnicas, antes de que sean realmente oportunos; además el productor debe tener un profundo respeto por las habilidades técnicas del director.

Es menos obvio que el productor pueda hacer todo tipo de cosas sutiles con los símbolos de estatus (el tamaño de la oficina, la alfombra, el mobiliario, las copias al carbón, etc.) para proclamar que el director, aunque fuera de la línea de gestión, es una fuente de poder de decisión.

Esto puede hacerse para trabajar de forma muy eficaz. Desafortunadamente rara vez se intenta. Los gestores de proyecto hacen mal su trabajo al utilizar al personal con talento técnico que no tiene cualidades sólidas para la gestión.

El director puede ser el jefe, y el productor su mano derecha. Robert Heinlein, en *El Hombre Que Vendió la Luna*, describe dicho acuerdo con un ejemplo ilustrativo.

> *Coster enterró su rostro en sus manos, luego alzó la vista. "Lo sabía, sabía lo que se necesitaba hacer – pero cada vez que trataba de enfrentar un problema técnico un imbécil quería que tomara decisiones acerca de los camiones –o teléfonos– o alguna maldita cosa. Lo siento, Sr. Harriman. Pensé que podía hacerlo."*
>
> *Harriman dijo gentilmente. "No dejes que esto te deprima, Bob. ¿No has dormido mucho últimamente, verdad? Te diré lo que vamos a hacer– pondremos a cargo a Ferguson. Tomaré ese escritorio por unos cuantos días y construiré un espacio para protegerte contra tales cosas. Quiero que tu cerebro piense en vectores de reacción, en eficiencia del combustible y cálculo de esfuerzos, no en contratar camiones." Harriman se paró en la puerta, buscó alguien fuera de la oficina, y notó un hombre que parecía ser el encargado principal de la oficina. "Oiga, usted! Venga aquí."*
>
> *El hombre pareció mostrarse sorprendido, se levantó, se acercó a la puerta y*

dijo: "¿Si?"

"Quiero que aquel escritorio de la esquina y todas las cosas que están encima sean cambiadas a una oficina vacía en este piso, hágalo ya"

Supervisó el cambio de Coster y su otro escritorio en otra oficina, observó que el teléfono en la nueva oficina estaba desconectado, y, como una ocurrencia repentina, mandó a instalar también un sofá. "Instalaremos un proyector, una máquina de dibujo, un estante y otros accesorios esta misma noche" le dijo a Coster. "Hágame una lista de todo lo que necesite– para trabajos de ingeniería" Regresó a la oficina nominal del jefe de ingeniería y se puso contento tratando de descubrir donde estaba parada la organización y lo que estaba mal en ella.

Unas cuatro horas después acompañó a Berkeley a encontrarse con Coster. El ingeniero en jefe estaba dormido en su escritorio, con la cabeza apoyada en los brazos. Harriman se disponía a marcharse y regresar, pero Coster despertó. "Lo siento" dijo, ruborizado, "Debí quedarme dormido."

"Este es el motivo de porqué te traje un sofá," dijo Harriman. "Es más confortable. Bob le presentó a Jock Berkeley. Es tu nuevo esclavo. Usted continuará como ingeniero en jefe y punto, jefe indiscutible. Jock será el Jefe Mayor de Todo lo Demás. Desde ahora no tienes que preocuparte por nada–excepto por el pequeño detalle de construir una nave Lunar."

Se estrecharon las manos. "Solo una pregunta, Sr. Coster", Dijo Berkeley seriamente, "páseme todo lo que quiera – usted dirigirá las cuestiones técnicas – pero por el amor de dios grábelo para que yo sepa qué ocurre. Tendrá un interruptor en su escritorio que operará una grabadora hermética en mi escritorio."

"Perfecto!" Coster ya se ve más joven, pensó Harriman.

"Y si usted desea algo que no sea técnico, no lo haga. Solo mueva el interruptor y silve; y será hecho!" Berkeley dirigió su mirada a Harriman. "El jefe quiere hablar contigo acerca del trabajo importante. Los dejo y me pondré manos a la obra." Y se marchó.

Harriman se sentó; Coster hizo lo mismo y dijo, "Waw!"

"¿Se siente mejor?"

"Me ha causado buena impresión ese Berkeley."

"Tanto mejor; desde ahora él es tu hermano gemelo. Deja de preocuparte; Trabajé antes con él. Pensarás que estás viviendo en un hospital bien acondicionado."[2]

Este relato apenas requiere análisis. Se puede hacer también este arreglo para trabajar de forma eficaz.

Sospecho que el último arreglo es mejor para equipos pequeños, como se discutió en el Capítulo 3, "El Equipo Quirúrgico." Creo que el arreglo más adecuado es poner al productor como jefe para los subárboles más grandes de un proyecto realmente grande.

La Torre de Babel fue quizá el primer fiasco de ingeniería, pero no fue el último. La comunicación y su consecuencia, la organización, son cruciales para el éxito. Las técnicas de comunicación y organización exigen del gestor mucha reflexión y tanta experiencia como la propia tecnología de software.

8
Predecir la Jugada

8
Predecir la Jugada

La práctica es el mejor de los maestros

PUBLILIUS

La experiencia es un buen maestro, pero los tontos no aprenderán de nadie.

ALMANAQUE DEL POBRE RICHARD

Douglass Crockwell, "Ruth predice su jugada", Serie Mundial, 1932
Reproducido con el permiso de Esquire Magazine y Douglas Crockwell, © 1945
(renovado en 1973) por Esquire, Inc., y la cortesía del Museo Nacional del Béisbol

¿Cuánto tiempo tomará un trabajo de programación de sistemas? ¿Cuánto esfuerzo será necesario? ¿Cómo se estima?

He sugerido anteriormente proporciones que al parecer se aplican al tiempo asignado a la planificación, codificación, prueba de componente y prueba del sistema. Primero, debo decir que *no* se puede estimar la tarea completa sólo estimando la parte de la codificación y luego aplicar las proporciones. La codificación representa sólo alrededor de una sexta parte del problema, y los errores en su estimación o en las proporciones podrían conducir a resultados ridículos.

Segundo, debo decir que los datos para construir programas aislados y pequeños no son aplicables a productos de los sistemas de programación. Por ejemplo, para un programa que en promedio tiene 3200 palabras, el informe de Sackman, Erickson, y Grant revela un tiempo promedio de codificación y depuración de aproximadamente 178 horas para un solo programador, una cifra que si fuera extrapolada daría una productividad anual de 35,800 declaraciones por año. Un programa de la mitad de ese tamaño tomó menos de una cuarta parte de tiempo, y si extrapolamos la productividad obtenemos casi 80,000 declaraciones por año.[1] Hay que añadir el tiempo invertido en la planificación, la documentación, las pruebas, el sistema de integración y el entrenamiento. La extrapolación lineal de tales cifras de velocidad no tiene sentido. La extrapolación de tiempos para la carrera de 100 yardas muestra que una persona puede correr una milla en menos de tres minutos.

Sin embargo, antes de descartar estas cifras, tengamos en cuenta que estos números, aunque no son para problemas estrictamente comparables, sugieren que el trabajo va como una potencia del tamaño *incluso* cuando no se involucra ningún tipo de comunicación excepto la de un hombre con sus recuerdos.

La Figura 8.1 nos cuenta la triste historia. Ilustra el resultado del informe de un estudio realizado por Nanus y Farr[2] en System Development Corporation. Muestra un exponente de 1.5; eso es,

$$\text{esfuerzo} = (\text{constante}) \times (\text{número de instrucciones})^{1.5}$$

Otro estudio de SDC reportado por Weinwurm[3] también muestra un exponente cercano a 1.5.

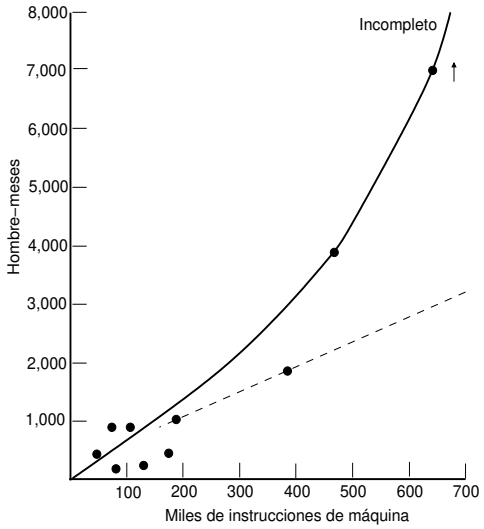


Fig. 8.1 Esfuerzo en función del tamaño del programa

Se han realizado unos cuantos estudios acerca de la productividad de los programadores, y se han propuesto varias técnicas de estimación. Morin ha preparado un estudio de los datos publicados.[4] Aquí mencionaré sólo unos cuantos puntos que parecen especialmente reveladores.

Los Datos de Portman

Charles Portman, gestor de la División de Software de ICL, en la Computer Equipment Organization (Northwest) en Manchester, ofrece otra útil visión personal.[5]

Encontró que sus equipos de programación erraban en el calendario por cerca de la mitad, pues cada trabajo demoraba aproximadamente el doble del tiempo estimado. Las estimaciones fueron realizadas de forma muy prolija por equipos experimentados que estimaron horas-hombre para centenares de subtareas en un diagrama PERT. Cuando apareció el patrón de retraso, les pidió mantener registros minuciosos del uso del tiempo. Estos mostraron que el error en la estimación podía ser totalmente atribuido al hecho de que sus

equipos solo estaban realizando el 50 por ciento del trabajo semanal con respecto al tiempo efectivo dedicado a la programación y a la depuración. El resto lo representaban el periodo de inactividad de la máquina, las breves tareas ajenas de mayor prioridad, las reuniones, el papeleo, los negocios de la compañía, la enfermedad, el tiempo dedicado a asuntos personales. En resumen, las estimaciones hicieron una suposición poco realista acerca del número de horas de trabajo técnico por hombre-año. Mi propia experiencia confirma bastante su conclusión.[6]

Los Datos de Aron

Joel Aron, gestor de Systems Technology en IBM en Gaithersburg, Maryland, ha estudiado la productividad del programador cuando trabajaba en nueve sistemas grandes (brevemente, *grande* significa más de 25 programadores y 30,000 instrucciones entregables).[7] Él divide tales sistemas de acuerdo al número de interacciones entre los programadores (y partes del sistema) y encuentra las siguientes productividades:

Muy pocas interacciones	10,000 instrucciones por hombre-año
Algunas interacciones	5,000
Muchas interacciones	1,500

Los hombre-años no incluyen tareas de apoyo y prueba del sistema, solo diseño y programación. Cuando estas cifras se alteran por un factor de dos para incluir la prueba del sistema, concuerdan mucho con los datos de Harr.

Los Datos de Harr

John Harr, gestor de programación de Electronic Switching System, de la Bell Telephon Laboratories informó acerca de su experiencia y la de otros en un artículo de 1969 en la Spring Joint Computer Conference.[8] Estos datos se muestran en las figuras 8.2, 8.3, y 8.4.

De estas, la Fig. 8.2 es la más detallada y útil. Las primeras dos tareas son básicamente programas de control; las otras dos son básicamente traductores de lenguajes. La productividad se expresa en términos de palabras depuradas por hombre-año. Esto incluye la programación, la prueba de componentes y la prueba del sistema. No está claro cuánto del esfuerzo en la planificación o

	Unids. de prog.	Número de programadores	Años	Hombre-años	Palabras de programa	Palabras/ hombre-año
Operacional	50	83	4	101	52,000	515
Mantenimiento	36	60	4	81	51,000	630
Compilador	13	9	$2\frac{1}{4}$	17	38,000	2230
Traductor (Ensamblador de Datos)	15	13	$2\frac{1}{2}$	11	25,000	2270

Fig 8.2 Resumen de las cuatro tareas No.1 de programación del ESS

del esfuerzo en apoyo de la máquina, la escritura y similares, se incluyeron.

Asimismo, las productividades caen en dos categorías; la de los programas de control que son de alrededor de 600 palabras por hombre-año; la de los traductores de cerca de 2200 palabras por hombre-año. Observe que los cuatro programas son de tamaño similar – la variación está en el tamaño de los grupos de trabajo, la duración y el número de módulos. ¿Cuál es la causa y cuál es el efecto? ¿Necesitaron los programas de control más personal porque fueron más complicados? ¿O necesitaron más módulos y más hombre-meses porque se les asignó más personal? No se puede estar seguro. Los programas de control seguramente fueron más complejos. Dejando de lado estas incertidumbres, los números describen las productividades reales llevadas a cabo en un sistema grande, usando las técnicas de programación actuales. Como tal, representan una contribución valiosa.

Las figuras 8.3 y 8.4 muestran algunos datos interesantes acerca de las tasas de programación y depuración comparadas con las tasas predichas.

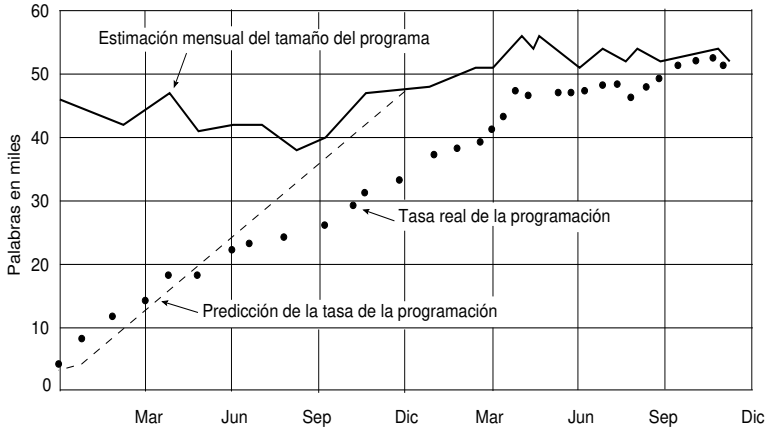


Fig. 8.3 Pronóstico del ESS y tasas de programación reales

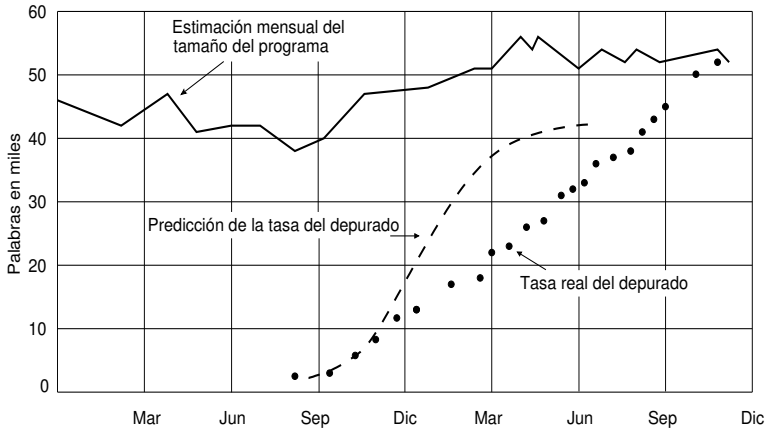


Fig. 8.4 Pronóstico del ESS y tasas de depuración reales

Los Datos del OS/360

La experiencia del IBM OS/360, aunque no está disponible con el detalle de los datos de Harr, lo confirma. Los grupos del programa de control experimentaron productividades en el rango de 600-800 instrucciones depuradas por hombre-año. Los grupos del traductor de lenguaje consiguieron productividades en el rango de 2000-3000 instrucciones depuradas por hombre-año. Estas incluyen la planificación realizada por el grupo, la codificación de la prueba del componente, la prueba del sistema y otras tareas de apoyo. Hasta donde puedo decir, son comparables a los datos de Harr.

Los datos de Aron, los de Harr y los del OS/360 confirman notables diferencias en la productividad relacionadas con la complejidad y dificultad de la tarea misma. Mi pauta en el embrollo de la estimación de la complejidad es que los compiladores resultan tres veces peores que los programas de aplicación por lotes normales, y los sistemas operativos resultan tres veces peores que los compiladores.[9]

Los Datos de Corbató

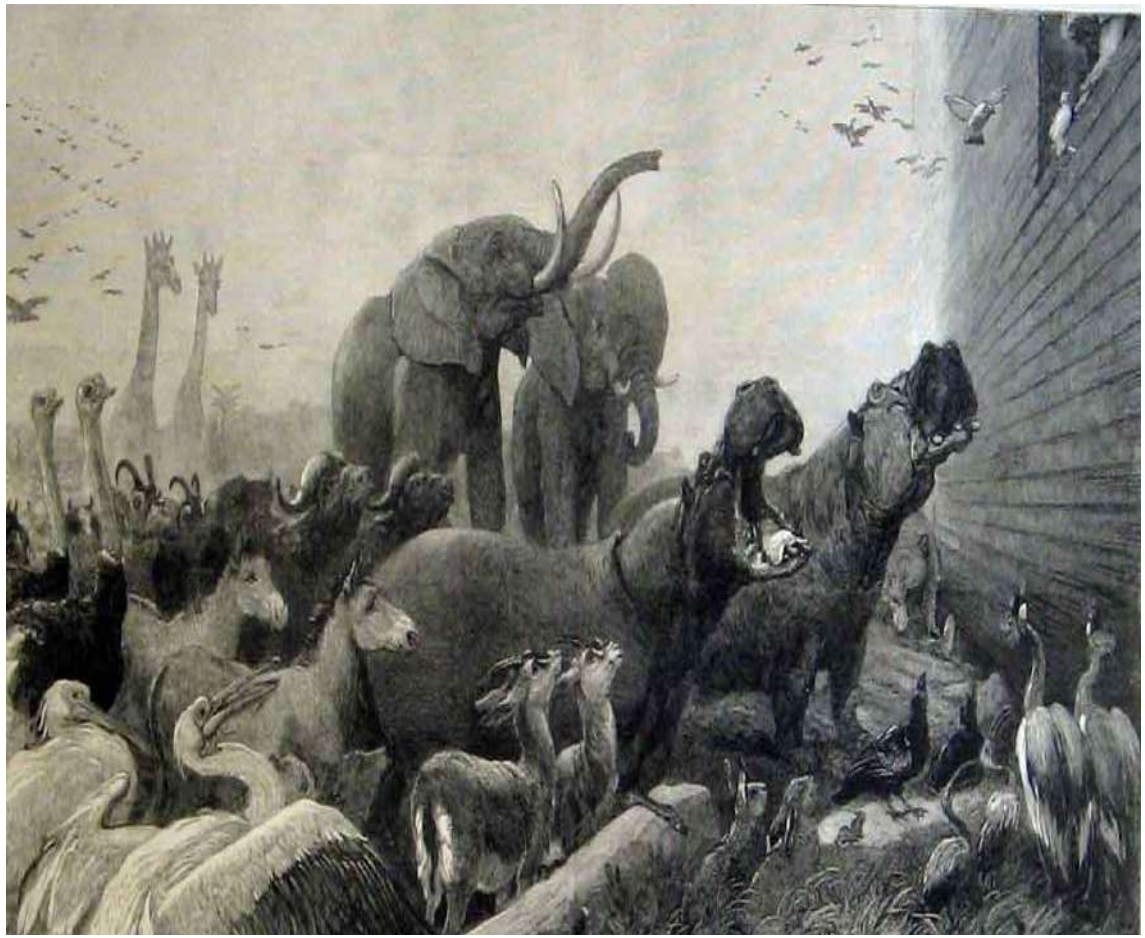
Tanto los datos de Harr como del OS/360 son para la programación en lenguaje ensamblador. Parece que se han publicado pocos datos acerca de la productividad en la programación de sistemas que usan lenguajes de alto nivel. Sin embargo, Corbató, del proyecto MAC del MIT reporta una productividad media de 1200 líneas de declaraciones depuradas en PL/I por hombre-año en el sistema MULTICS (entre 1 y 2 millones de palabras).[10]

Esta cifra es muy interesante. Al igual que los demás proyectos, MULTICS incluye programas de control y traductores de lenguaje. Como los otros, está produciendo un producto de programación de sistemas, probado y documentado. Los datos parecen ser comparables en términos del tipo de esfuerzo inplicado. Y la cifra de productividad es un buen promedio entre el programa de control y las productividades del traductor de otros proyectos.

Aunque la cifra de Corbató es *líneas* por hombre-año, no *palabras!*. Cada declaración en su sistema corresponde a alrededor de cinco palabras de código escrito a mano! Lo que sugiere dos conclusiones importantes:

- La productividad parece constante en términos de declaraciones elementales, una conclusión que es razonable en términos del esfuerzo mental que requiere una declaración y los errores que puede incluir.[11]
- La productividad de la programación se puede incrementar casi cinco veces cuando se usa un lenguaje de alto nivel adecuado.[12]

9
Diez Libras en un Costal de Cinco Libras



9
Diez Libras en un Costal de Cinco Libras

El autor debería hechar una mirada a Noé, y . . . aprender, cómo lograron abarrotar en el arca un montón de cosas en un espacio tan pequeño.

SYDNEY SMITH, EDIMBURGH REVIEW

Grabado de una pintura de Heywood Hardy
El archivo Bettman

Espacio del Programa como Costo

¿Cuán grande es? Aparte del tiempo de ejecución, el espacio de memoria que ocupa un programa representa un costo importante. Esto es cierto incluso para programas propietario, donde el usuario paga al autor una tarifa que es principalmente una porción del costo de desarrollo. Considere el sistema de software interactivo IBM APL. Se renta por $400 al mes y, cuando se usa, ocupa al menos 160 K bytes de memoria. En una Model 165, la memoria se renta a $12 por kilobyte al mes. Si se dispone del programa a tiempo completo, se paga $400 por la renta del software y $1920 por la renta de memoria por usar el programa. Si se usa el sistema APL sólo por cuatro horas al día, los costos son de $400 por la renta del software y $320 por la renta de memoria al mes.

Con frecuencia escuchamos expresiones de alarma porque una máquina de 2 M byte pueda dedicar 400 K a su sistema operativo. Esto es tan tonto como criticar a un Boeing 747 porque su costo es de $27 millones. Más bien debemos preguntarnos, "¿Qué es lo que hace?" ¿Qué obtenemos por los dólares invertidos en facilidad de uso y rendimiento (a través del uso eficiente del sistema)? ¿Podrían los $4800 por mes invertidos en renta de memoria haber sido gastados de manera más provechosa en otro hardware, en programadores o en programas de aplicación?

El diseñador del sistema asigna parte del total de sus recursos de hardware a memoria del programa residente cuando piensa que de esta forma hará más por el usuario que como sumadores, discos, etc. Hacerlo de otra manera sería extremadamente irresponsable. Además, el resultado debe juzgarse como un todo. Nadie puede criticar un sistema de programación por el tamaño *per se* y al mismo tiempo abogar constantemente por una integración más cercana entre el diseño del hardware y el software.

Puesto que el tamaño representa una porción tan grande del costo de uso de un producto de sistemas de programación, el constructor debe establecer metas respecto al tamaño, controlar el tamaño, e idear técnicas para reducir ese tamaño, así como el constructor de hardware establece metas en la cantidad de componentes, controla la cantidad de componentes, e idea técnicas de reducción de componentes. Como cualquier costo, el tamaño como tal no es malo, pero el tamaño innecesario sí lo es.

Control del Tamaño

Para el gestor de proyecto, el control del tamaño es en parte un trabajo técnico y en parte un trabajo de gestión. Se debe estudiar a los usuarios y sus aplicaciones para establecer las características del tamaño de los sistemas que serán ofrecidos. Después, hay que subdividir estos sistemas y asignar a cada componente un determinado tamaño. Puesto que los compromisos entre tamaño y velocidad vienen en saltos cuánticos bastante grandes, establecer los objetivos del tamaño es una cuestión delicada, que requiere conocimiento de los compromisos alcanzables dentro de cada pieza. El gestor prudente también debe reservar un fondo, que será asignado a medida que el trabajo avanza.

En el OS/360, aunque todo esto se hizo de forma prolija, aún tuvimos que aprender otras dolorosas lecciones.

Primero, establecer metas respecto al tamaño del espacio de memoria del núcleo no es suficiente; se tienen que fijar todos los aspectos relativos al tamaño. En la mayoría de los sistemas operativos anteriores, los sistemas residían en cinta, y no se intentaba usarlos descuidadamente para incorporar segmentos de programa debido a los largos tiempos de búsqueda. El OS/360 residía en disco, como sus predecesores inmediatos, el Stretch Operating System y el 1410-7019 Disk Operating System. Sus constructores se alegraron por la libertad de los módicos accesos a disco. El resultado inicial fue un rendimiento desastroso.

Cuando asignamos porciones del núcleo a cada componente, no fijamos simultáneamente los costos de los tiempos de acceso. Como podría esperar cualquiera que lo viera en retrospectiva, un programador encontraba a su programa volcado en su núcleo objetivo desmenuzado en superposiciones. Este proceso en sí mismo no solo aumentó el tamaño total sino tambien ralentizó la ejecución. Y pero aún, nuestro sistema de control de gestión ni media ni se percataba de esto. Todos informaban acerca de la cantidad de *núcleo* que estaban usando, y puesto que estaban dentro de lo establecido, nadie se preocupaba

Afortunadamente, pronto llegó un día en el trabajo en que el simulador de rendimiento del OS/360 empezó a funcionar. El primer resultado indicaba un grave problema. El Fortran H, una Model 65 con memoria de tambor, simulaba la compilación a una razón de cinco declaraciones por minuto! La investigación mostró que los módulos del programa de control estaban rea-lizando

demasiados accesos a disco. Incluso los módulos supervisores de alta frecuencia estaban haciendo muchos viajes al pozo, y el resultado fue algo bastante parecido a una hiperpaginación.

La primera moraleja es clara: Establecer el tamaño *total* del espacio de memoria como también el tamaño del espacio de memoria residente; establecer los accesos al almacenamiento de respaldo así como también su respectivo tamaño.

La siguiente lección fue muy similar. La cantidad de espacio se fijó antes de hacer las asignaciones funcionales exactas a cada módulo. El resultado, cualquier programador con problemas de espacio examinaba su código para ver qué podía arrojar sobre la cerca al espacio del vecino. Así que la memoria intermedia administrada por el programa de control llegó a ser parte del espacio del usuario. Más grave aún, así se hizo con todo tipo de bloques de control, y el efecto fue comprometer completamente la seguridad y la protección del sistema.

Así pues, la segunda moraleja también es clara: Definir exactamente qué debe hacer cada módulo cuando se especifique cuán grande debe ser.

A través de estas experiencias se expone una tercera y más profunda lección. El proyecto era suficientemente grande y la gestión de comunicación suficientemente mala para provocar que muchos miembros del equipo se consideraran concursantes por ganar premios, en lugar de constructores de productos de programación. Todos suboptimizaban sus piezas por cumplir sus objetivos; unos cuantos se detenían a pensar acerca del efecto total en el cliente. Esta falla en la orientación y la comunicación es un gran peligro para proyectos grandes. Durante toda la implementación, los arquitectos del sistema deben mantener una vigilancia constante para asegurar una integridad perdurable del sistema. Sin embargo, más allá de este mecanismo de vigilancia yace la cuestión de la actitud de los implementadores mismos. Fomentar una actitud de sistema total, y orientado al usuario bien podría ser la función más importante del gestor de programación.

Técnicas de espacio

No hay cantidad ni control de espacio de memoria asignado que pueda lograr que un programa sea pequeño. Eso requiere ingenio y destreza.

Es obvio que una mayor funcionalidad implica un mayor espacio, manteniendo la velocidad constante. Así pues, el primer campo de la destreza está en el compromiso entre funcionalidad y tamaño. Aquí surge una antigua y profunda pregunta de política. ¿Cuánto de dicha elección se debe reservar al usuario? Se puede diseñar un programa con muchos rasgos opcionales, cada uno de los cuales ocupa un poco de espacio. Se puede diseñar un generador que tome una lista de opciones y confeccione un programa para ella. Aunque para cualquier conjunto particular de opciones, un programa más monolítico ocuparía menos espacio. Es muy parecido a un auto; si la luz de los mapas, el encendedor y el reloj se cotizan juntos como una sola opción, el paquete costará menos que si se pudiera escoger cada una de forma separada. Por lo tanto, el diseñador debe decidir cuánto grano fino deberá tener la elección de opciones del usuario.

Al diseñar un sistema para una gama de tamaños de memoria, surge otra cuestión primordial. Hay un efecto limitador que impide que el rango idóneo se extienda arbitrariamente, incluso con la modularidad de grano fino de la funcionalidad. En el sistema más pequeño, la mayoría de los módulos estarán superpuestos. Por tanto, se debe reservar una porción importante del espacio de memoria residente del sistema más pequeño como un área temporal o un área de paginado en el cual se busquen otras secciones. Su tamaño determinará el tamaño de todos los demás módulos. Además, el dividir funciones en módulos pequeños tiene un costo tanto en espacio como en rendimiento. Así que un sistema grande que puede permitirse un área temporal veinte veces mayor, de este modo solo se ahorrará en accesos. Aún padecerá de velocidad y de espacio, pues el tamaño del módulo es muy pequeño. Este efecto limita la máxima eficiencia del sistema que se puede generar a partir de módulos de un sistema pequeño.

El segundo campo de la destreza son los compromisos espacio-temporales. Dadas cierta funcionalidad, mayor espacio de memoria implica mayor velocidad. Esto es cierto en un rango impresionantemente extenso. Este hecho es el que hace posible establecer la cantidad de espacio de memoria.

El gestor puede hacer dos cosas para ayudar a su equipo a realizar buenos compromisos entre espacio y tiempo. Una es asegurarse de que están capacitados en técnicas de programación, no solo dejarlos depender del ingenio natural y la experiencia. Esto es especialmente importante para un nuevo

lenguaje o una nueva máquina. Se debe aprender rápidamente los detalles para usarlos con destreza y compartirlos ampliamente, quizás con premios especiales o elogios por las nuevas técnicas.

La segunda es reconocer que la programación tiene una tecnología, y los componentes deben *ser* fabricados. Cada proyecto debe tener un cuaderno lleno de buenas subrutinas o macros para el manejo de colas, búsqueda, dispersión y clasificación. Para cada una de tales funciones el cuaderno debe tener al menos dos programas, la rápida y la comprimida. El desarrollo de dicha tecnología es una labor importante de comprensión que puede llevarse a cabo en paralelo con la arquitectura del sistema.

La Representación es la Esencia de la Programación

Mas allá de la destreza yace la invención, y es aquí donde nacen los programas ligeros, sobrios, y rápidos. Estos son, casi siempre, el resultado de adelantos estratégicos en lugar de tácticas inteligentes. A veces el adelanto estratégico será un nuevo algoritmo, tal como la Transformada Rápida de Fourier de Cooley-Turkey o la sustitución de una clasificación $n\,log(n)$ por un conjunto de n^2 comparaciones.

Es mucho más frecuente que el adelanto estratégico venga de rehacer la representación de los datos o tablas. Aquí es donde yace el corazón de un programa. Enséñeme sus diagramas de flujo y oculte sus tablas, y seguiré desconcertado. Muéstreme sus tablas, y generalmente no necesitaré sus diagramas de flujo; serán obvios.

Es fácil reproducir ejemplos del poder de las representaciones. Recuerdo a un joven emprendiendo la construcción de un complejo intérprete de consola para una IBM 650. Como el espacio de memoria era apreciado, terminó empaquetándolo en una increíblemente pequeña cantidad de espacio al construir un intérprete para un intérprete, identificando que las interacciones humanas son lentas e infrecuentes. El pequeño y elegante compilador de Fortran de Digitek usa una representación especializada y muy densa para el propio código del compilador, por lo que no se necesita almacenamiento externo. El tiempo perdido en la decodificación de esta representación se recupera diez veces al evitar entradas y salidas. (Los ejercicios al final del Capítulo 6 del libro *Automatic Data Processing*[1] de Brooks e Iverson, incluyen una colección de

tales ejemplos, al igual que muchos ejercicios de Knuth.[2]).

El programador al borde de la desesperación debido a la carencia de espacio de memoria a menudo lo mejor que puede hacer es desatenderse de su código, voltear, y contemplar sus datos. La representación *es* la esencia de la programación.

10
La Hipótesis Documental

10
La Hipótesis Documental

La hipótesis:

Inundado en papeles, un puñado de documentos se convirtieron en los ejes centrales alrededor de los cuales gira toda gestión de proyectos. Representan las principales herramientas personales del gestor.

W. Bengough, "Escena en la vieja Biblioteca del Congreso," 1897
El archivo Bettman

La tecnología, el entorno organizativo, así como las tradiciones del oficio, todos ellos intervienen en la elaboración de ciertos elementos de la documentación que todo proyecto debe elaborar. Para el nuevo gestor, fresco aún de fungir él mismo como artesano, todos estos documentos le parecen una absoluta molestia, una distracción innecesaria, una marea blanca que amenaza con engullirlo. Y en efecto, la mayoría de ellos son exactamente eso.

Sin embargo, poco a poco se da cuenta que un puñado de estos documentos representa y expresa gran parte de su trabajo de gestión. La preparación de cada uno sirve como una oportunidad para concentrar las ideas y plasmar las discusiones que de otra manera vagarían sin fin. Su mantenimiento llega a ser su mecanismo de supervisión y alerta. El documento en sí sirve como una lista de control, un control del estado y una base de datos para sus informes.

Veamos cómo debería funcionar esto en un proyecto de programación, examinemos la utilidad de determinados documentos en otros contextos y ver si surge una generalización.

Documentos para un Producto Informático

Supongamos que estamos construyendo una computadora. ¿Qué documentos son fundamentales?

Los objetivos. Definen las necesidades que se deben cumplir y las metas, los propósitos, las restricciones y las prioridades.

Las especificaciones. Es el manual de la computadora más las especificaciones de rendimiento. Es uno de los primeros documentos que se elaboran en la propuesta de un nuevo producto, y el último en terminarse.

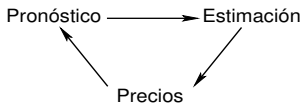
Calendario

Presupuesto. El presupuesto no es únicamente una restricción, sino uno de los documentos más útiles del gestor. La existencia del presupuesto obliga a tomar decisiones técnicas que de otra manera se evitarían; y es más, aclara y obliga a tomar decisiones de política.

Organigrama

Asignaciones de espacio

Estimación, pronóstico y precios. Estos tres se entrelazan de forma cíclica, y determinan el éxito o fracaso del proyecto:



Para generar un pronóstico de mercado, uno requiere especificaciones de rendimiento y precios tentativos. Las cifras del pronóstico combinadas con la suma de componentes del diseño determinan la estimación en el costo de fabricación, y determinan la contribución por unidad de desarrollo y los costos fijos. Estos costos, a su vez, determinan los precios.

Si los precios están *por debajo* de los propuestos, empieza una afortunada espiral exitosa. Las expectativas aumentan, los costos por unidad caen, y los precios caen aún más.

Si los precios están *por encima* de los propuestos, empieza una espiral desastrosa, y todos deben luchar por quebrantarla. Debemos sacarle jugo al rendimiento y desarrollar nuevas aplicaciones para apoyar pronósticos mayores. Reducir costos para producir estimaciones menores. La tensión de este ciclo es un proceso que con frecuencia evoca el mejor trabajo de vendedores e ingenieros.

Esto también puede provocar irresoluciones absurdas. Recuerdo una máquina cuyo contador de instrucciones brincaba dentro o fuera de la memoria cada seis meses en el transcurso de un ciclo de desarrollo de tres años. En una fase se buscaba un poco más de rendimiento, así que el contador de instrucciones se implementaba con transistores. En la siguiente fase el tema era la reducción de costos, así que el contador se implementaba como una localidad de memoria. En otro proyecto, el mejor gestor de ingeniería que he conocido, con frecuencia funcionaba como un volante inmenso, cuya inercia amortiguaba las fluctuaciones que venían del mercado y de los gestores.

Documentos para un Departamento Universitario

A pesar de las inmensas diferencias en el propósito y quehacer, un número similar de documentos similares conforman el conjunto de documentos importantes para el presidente de un departamento universitario. Casi toda de-

cisión que toma el decano, la reunión de profesores, o el presidente es una especificación o cambio en los siguientes documentos:

Objetivos
Descripción del curso
Requisitos de titulación
Propuestas de investigación (por lo tanto planes, cuando se financian)
Horario de clases y tareas docentes
Presupuesto
Asignación de espacio
Asignación de personal y estudiantes de posgrado

Observe que los componentes son muy parecidos a los de un proyecto informático: objetivos, especificaciones del producto, asignaciones de tiempo, asignaciones de dinero, asignaciones de espacio y asignaciones de personal. Solo faltan los documentos relativos al precio; aquí es donde la legislatura hace su trabajo. Las semejanzas no son accidentales – los asuntos de cualquier tarea de gestión son: qué, cuándo, cuánto, dónde y quién.

Documentos para un Proyecto de Programación

En muchos proyectos de programación, el personal empieza a participar en reuniones para discutir la estructura; después empiezan a escribir programas. Sin embargo, no importa cuán pequeño sea el proyecto, es buena idea que el gestor empiece inmediatamente a formalizar al menos mini-documentos que sean su base de datos. Pues terminará necesitando documentos muy parecidos a los de otros gestores.

Qué: objetivos. Define la necesidad que se debe cumplir y las metas, los propósitos, las restricciones y las prioridades.

Qué: especificaciones del producto. Empieza como una propuesta y termina como el manual y la documentación interna. Las especificaciones de velocidad y espacio son una parte crucial.

Cuándo: calendario

Cuánto: presupuesto

Dónde: asignación de espacio

Quién: organigrama. Esto se entrelaza con la especificación de la interfaz, como predice le ley de Conway: "Las organizaciones que diseñan sistemas están obligadas a producir sistemas que son copias de las estructuras de comunicación de tales organizaciones."[1] Conway señala que el organigrama reflejará inicialmente el diseño del primer sistema, que casi con seguridad no será el correcto. Si el diseño del sistema es libre de cambiar, la organización debe estar preparada también para hacerlo.

¿Por Qué Tener Documentos Formales?

Primero, es esencial anotar las decisiones. Solo cuando se escribe surgen los vacíos y sobresalen las contradicciones. El acto de escribir prueba que se requieren cientos de mini-decisiones y es la existencia de éstas las que distinguen las políticas claras y exactas de las difusas.

Segundo, los documentos comunicarán las decisiones a los demás. Al gestor le sorprenderá muy a menudo darse cuenta de que las políticas que adoptó para el conocimiento de todos son totalmente desconocidas por algunos miembros de su equipo. Puesto que su trabajo primordial es mantener a todos avanzando en la misma dirección, su principal tarea diaria será la comunicación, no la toma de decisiones, y sus documentos aliviarán enormemente esta carga.

Finalmente, los documentos del gestor le dan una base datos y una lista de control. Al revisarlos periódicamente observa dónde está, y observa qué cambios de énfasis o cambios de dirección son necesarios.

No comparto el punto de vista proyectado por el vendedor del "sistema de información total de gestión", en donde el ejecutivo realiza una consulta en la computadora, y una pantalla despliega su respuesta. Existen muchas razones importantes por las que nunca sucederá esto. Un motivo es que solo una pequeña porción – quizá el 20 por ciento – del tiempo del ejecutivo se invierte en tareas donde necesita información externa. El resto es comunicación: escuchar, informar, enseñar, exhortar, conciliar, animar. Pero para

esa fracción que depende de datos, el puñado de documentos cruciales son vitales, y satisfarán casi todas sus necesidades.

La tarea del gestor es desarrollar un plan y luego llevarlo a cabo. Pero solo el plan escrito es preciso y comunicable. Dicho plan consta de documentos acerca de qué, cuándo, cuánto, dónde y quién. Este pequeño conjunto crucial de documentos encapsula gran parte del trabajo del gestor. Si se reconoce, desde un principio, su naturaleza exhaustiva y crítica, el gestor puede aprovecharlos como herramientas amigables en lugar de una fastidiosa tarea inútil. Haciendo esto se encaminará de forma más prolija y rápida.

11
Planifique
Desechar

11
Planifique
Desechar

No hay nada en este mundo constante salvo la inconstancia

SWIFT

Es de sentido común tomar un método y probarlo. Si falla, admitirlo sinceramente y probar otro. Pero sobretodo probar alguno.

FRANKLIN D. ROOSEVELT

Colapso del aerodinámicamente mal diseñado Tacoma Narrows Bridge, 1940
Foto UPI/El Archivo Bettman

Plantas Piloto y Ampliación

Los ingenieros químicos han aprendido desde hace mucho tiempo que un proceso que funciona en laboratorio no es posible implementarlo en una fábrica en un solo paso. Se requiere un paso intermedio conocido como *planta piloto* para experimentar con cantidades a mayor escala y operando en ambientes no protegidos. Por ejemplo, un proceso de laboratorio para desalinizar agua debe ser probado en una planta piloto con capacidad de $10,000$ *galones/día* antes de ser usado en un sistema de agua comunitario de $2,000,000$ *galones/día*.

Los constructores de sistemas de programación también se han expuesto a esta lección, aunque al parecer todavía no han aprendido. Proyecto tras otro se diseña un conjunto de algoritmos y luego se sumerge en la construcción del software para su entrega al cliente en base a un calendario que exige la entrega de lo primero que se construye.

En la mayoría de los proyectos, el primer sistema construido apenas es útil. Pues suele ser muy lento, muy grande, o incómodo de usar, o todos juntos. No hay opción sino empezar de nuevo, dolido pero con más experiencia, y construir una versión rediseñada en la cual se resuelvan estos problemas. El descartar y rediseñar se puede realizar en un solo bloque, o pieza por pieza. Toda la experiencia en sistemas grandes muestra que de cualquier manera eso se hará.[2] Cuando se usa un nuevo concepto de sistema o una nueva tecnología, se tiene que construir un sistema para desecharlo, pues incluso la mejor planificación no es tan omnisciente como para obtener el sistema correcto la primera vez.

Por lo tanto, el dilema del gestor no es *si* construir un sistema piloto y desecharlo. Eso se *hará* de cualquier manera. El único dilema es si se va a planificar por adelantado construir un sistema desechable, o prometer entregar el sistema desechable a los clientes. Visto de esta manera, la respuesta es mucho más clara. Entregar el sistema desechable a los clientes compra tiempo, aunque sólo a costa de la angustia del usuario, la distracción de los constructores mientras realizan el rediseño, y una mala reputación del producto que el mejor rediseño resultará difícil de superar.

Así que *planifique desechar; lo hará de cualquier modo.*

Lo Único Constante Es el Cambio Mismo

Una vez que se acepta que un sistema piloto se va a construir y descartar, y que es inevitable un rediseño con ideas modificadas, resulta útil encarar todo el fenómeno del cambio. El primer paso es aceptar el hecho del cambio como forma de vida, en lugar de una excepción inadecuada y molesta. Cosgrove ha señalado perspicazmente que un programador entrega satisfacción a una necesidad del usuario en lugar de un producto tangible. Y tanto la necesidad real como la percepción del usuario de esa necesidad cambiarán a medida que se construyan, prueben y usen los programas.[3]

Por supuesto, esto también se aplica a las necesidades que deben satisfacer los productos de hardware, ya sean autos nuevos o computadoras nuevas. Aunque la propia existencia de un objeto tangible sirve para contener y cuantificar la demanda del usuario por cambios. Tanto la ductibilidad como la invisibilidad del producto del software exponen a sus constructores a cambios perpetuos en los requisitos.

Lejos estoy de sugerir que todos los cambios en los objetivos y requisitos del cliente deben, pueden, o debieran ser incorporados en el diseño. Evidentemente se debe fijar un umbral, y debe ser cada vez mayor a medida que avanza el desarrollo, o jamás surgirá un producto.

Sin embargo, algunos cambios en los objetivos son inevitables, y es mejor estar preparados para ellos que suponer que nunca llegarán. No sólo son inevitables los cambios en los objetivos, también son inevitables los cambios en la estrategia de desarrollo y la técnica. El concepto de desechar es por sí mismo sólo una aceptación del hecho de que a medida que se aprende, se cambia el diseño.[4]

Planifique el Sistema para el Cambio

Las formas de diseñar un sistema para tales cambios son bastante conocidas y ampliamente discutidas en la literatura – quizá más ampliamente discutidas que practicadas. Incluyen una modularización exhaustiva, un amplio uso de subrutinas, una definición precisa y completa de interfaces entre módulos, y una documentación completa de éstas. Es menos evidente que se quiera usar, donde sea posible, secuencias de llamadas estándar y técnicas manejadas por tablas.

Es más importante el uso de lenguajes de alto nivel y técnicas de autodocumentación para reducir errores inducidos por los cambios. Un auxiliar poderoso para realizar cambios es el uso de operaciones en tiempo de compilación para incorporar declaraciones estándar.

Una técnica fundamental es la cuantificación del cambio. Cada producto debería tener versiones numeradas, y cada versión debe tener su propio calendario y fecha de congelación, después de la cual los cambios entran en la siguiente versión.

Planifique la Organización para el Cambio

Cosgrove aboga por tratar todos lo planes, hitos y calendarios como provisionales, para facilitar así los cambios. Esto va demasiado lejos, pues actualmente la falla común de los grupos de programación es el muy poco control de gestión, no el exagerado.

Sin embargo, Cosgrove ofrece importantes revelaciones. Observa que la reticencia a documentar el diseño no se debe únicamente a la pereza o a presiones de tiempo. Se debe en cambio a la reticencia del diseñador a comprometerse en la defensa de decisiones que sabe que son provisionales. "A través de la documentación de un diseño, el diseñador se expone a la crítica de todos, y debe ser capaz de defender todo lo que escribe. Si existe alguna amenaza de la estructura organizativa, nada será documentado hasta que sea completamente defendible."

Estructurar una organización para el cambio es mucho más difícil que diseñar un sistema para el cambio. Toda persona debe ser asignada a trabajos que lo amplíen, tal que toda la fuerza de trabajo sea técnicamente flexible. En proyectos grandes el gestor debe mantener a dos o a tres de los mejores programadores como caballería técnica que pueda galopar al rescate donde sea que la batalla sea más tupida.

Mientras el sistema cambia también se deben cambiar las estructuras de gestión. Esto significa que el jefe debe prestar mucha atención a conservar a sus gestores y a su personal técnico como intercambiables en tanto sus aptitudes lo permitan.

Existen barreras sociológicas, y deben ser enfrentadas con una vigilancia constante. Primero, los propios gestores piensan del personal con experiencia como "muy valiosos" para usarlos en la programación real. Luego, el tra-

bajo de gestión conlleva mayor prestigio. Para superar este problema algunos laboratorios, tales como Bell Labs, abolieron todos los nombres de los puestos de trabajo. Todo empleado profesional es un "miembro del personal técnico." Otros, como IBM, mantienen una doble jerarquía de ascensos, como muestra la Fig. 11.1. Los peldaños correspondientes en teoría son equivalentes.

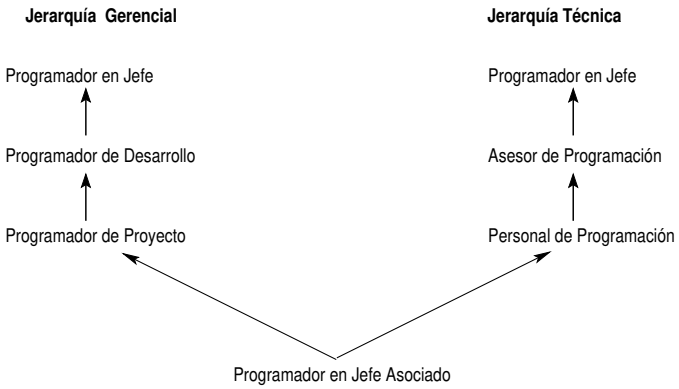


Fig. 11.1 Jerarquía dual de ascensos de IBM

Es fácil fijar escalas salariales correspondientes a cada peldaño. Es mucho más difícil darles el prestigio correspondiente. Las oficinas tienen que ser del mismo tamaño y cargo. Los servicios secretariales y otros servicios de apoyo deben ser equivalentes. Una reasignación de la jerarquía técnica a un nivel gerencial correspondiente jamás debe ir acompañada de un aumento, y debe comunicarse siempre como una "reasignación" nunca como un "ascenso." La reasignación inversa siempre debe llevar un aumento, es necesario sobrecompensar a causa de las presiones culturales.

Los gestores deben ser enviados a cursos de actualización técnica, el personal técnico con experiencia a entrenamiento en gestión. Los objetivos del

proyecto, los avances, y los problemas de gestión deben ser compartidos con todo el cuerpo del personal con experiencia.

Siempre y cuando las aptitudes lo permitan, el personal con experiencia debe mantenerse listo técnica y emocionalmente para gestionar grupos o para disfrutar construyendo programas con sus propias manos. Con seguridad hacer esto requiere mucho trabajo; pero con seguridad lo vale!

Toda esa noción de organizar a los equipos de programación como un equipo quirúrgico es un ataque radical a este problema. Tiene el efecto de lograr que una persona experimentada no se sienta menospreciada cuando desarrolle programas, así se intenta eliminar los obstáculos sociales que lo privan del placer creativo.

Además, esa estructura está diseñada para minimizar el número de interfaces. Como tal, hace que el sistema cambie con máxima facilidad, y llega a ser relativamente fácil reasignar todo un equipo quirúrgico a una tarea diferente de programación cuando se requieren cambios organizativos. Es realmente la respuesta a largo plazo al problema de la organización flexible.

Dos Pasos Adelante y Uno Atrás

Un programa no deja de cambiar cuando se libera para uso del cliente. Los cambios después de la liberación se llaman *mantenimiento del programa,* aunque el proceso es esencialmente diferente al mantenimiento del hardware.

El mantenimiento del hardware para un sistema computacional involucra tres tareas – reemplazar los componentes deteriorados, limpiar y lubricar, e introducir cambios de ingeniería que corrijan los defectos de diseño. (La mayor parte de los cambios de ingeniería, aunque no todos, corrijen defectos en la ejecución o implementación, en lugar de defectos en la arquitectura, así que son invisibles para el usuario.)

El mantenimiento del programa no involucra limpieza, lubricación, o reparación del deterioro. Consiste principalmente de cambios que corrijen los defectos de diseño. Estos cambios, tienen que ver, con mayor frecuencia, con la adición de funcionalidades que con el hardware. Y generalmente son visibles para el usuario.

El costo total de mantenimiento de un programa ampliamente usado representa normalmente el 40 por ciento o más del costo de desarrollo. Es sor-

prendente, pero este costo está afectado principalmente por el número de usuarios. Más usuarios encuentran más errores.

Betty Campbell, del Laboratorio de Ciencias Nucleares del MIT, muestra un ciclo interesante en la vida de una liberación específica de un programa. Se muestra en la Fig. 11.2. Al inicio, los antiguos errores encontrados y corregidos en liberaciones anteriores tienden a reaparecer en una nueva versión. Las nuevas funcionalidades de la nueva liberación resultan tener defectos. Se remueven estas cosas, y todo parece ir bien por varios meses. Luego la tasa de errores empieza a emerger de nuevo. La señorita Campbell piensa que esto se debe al arribo de usuarios a una nueva plataforma de sofisticación, donde ellos empiezan a operar plenamente las nuevas capacidades de la liberación. Entonces, este ejercicio intenso pone al descubierto los errores más sutiles en las nuevas características.[5]

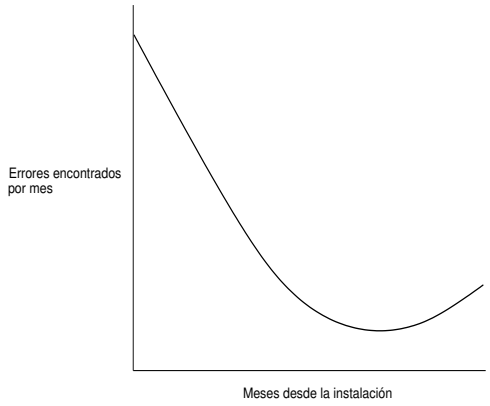


Fig. 11.2 Aparición de errores en función de la edad de la liberación

El problema esencial con el mantenimiento de programas es que corregir un defecto tiene una considerable probabilidad (20-50 por ciento) de introducir otro. Así que todo el proceso está dos pasos adelante y uno atrás.

¿Por qué los defectos no se corrigen con mayor limpieza? Primero, incluso un defecto sutil se muestra como una avería local de cierto tipo. De hecho, con frecuencia tiene ramificaciones que abarcan todo el sistema, y generalmente no son obvias. Cualquier intento por corregirlo con el mínimo esfuerzo corregirá el error local y obvio, pero a menos que la estructura sea pulcra o la documentación muy buena, los efectos de gran alcance de la corrección se pasarán por alto. Segundo, la persona que repara generalmente no es la que escribió el código, y con frecuencia es un programador sin experiencia o un aprendiz.

Como consecuencia de la introducción de nuevos errores, el mantenimiento del programa requiere muchas más pruebas del sistema por declaración escrita que cualquier otra etapa de la programación. En teoría, después de cada corrección se debe ejecutar el banco entero de casos de prueba previamente ejecutados en contra del sistema, para garantizar que no se haya dañado de una forma desconocida. En la práctica tales *pruebas de regresión*, de hecho, deben aproximarse a este ideal teórico, y eso es muy costoso.

Es claro que los métodos de diseño de programas para eliminar o al menos destacar los efectos laterales pueden tener un inmenso beneficio en los costos de mantenimiento. Así también pueden hacerlo los métodos de implementación de diseños con menos personas, menos interfaces y por lo tanto menos errores.

Un Paso Adelante y Otro Atrás

Lehman y Belady estudiaron la historia de sucesivas liberaciones de un sistema operativo grande.[6] Descubrieron que la cantidad total de módulos se incrementa linealmente con el número de liberación, pero que la cantidad de módulos afectados se incrementa exponencialmente con el número de liberación. Todas las correcciones tienden a destruir la estructura, a incrementar la entropía y el desorden del sistema. Cada vez se invierte menos esfuerzo en arreglar los defectos originales de diseño; y mayor esfuerzo en reparar los defectos introducidos por correcciones anteriores. Conforme pasa el tiempo, el sistema se vuelve cada vez menos bien-ordenado. Tarde o temprano las

reparaciones cesan de ganar terreno. Cada paso hacia adelante se iguala con otro hacia atrás. El sistema, aunque en principio es operable por siempre, se ha agotado como base para el progreso. Además, las máquinas cambian, las configuraciones cambian, y los requisitos del usuario también cambian, por lo que el sistema no es operable eternamente. Es necesario un flamante rediseño partiendo desde cero.

Belady y Lehman, partiendo de un modelo de la mecánica estadística llegaron a una conclusión general para los sistemas informáticos basados en la experiencia de todo el planeta. Pascal afirmaba, "Las cosas siempre están mejor al principio". C.S. Lewis lo ha expresado con más perspicacia:

> *Esta es la clave de la historia. Se invierte una tremenda energía – se construyen las civilizaciones – se idean excelentes instituciones; pero siempre algo sale mal. Algún defecto fatal siempre lleva a la gente egoista y cruel al poder, y luego todo cae de nuevo en la ruina y la miseria. De hecho, la máquina se estropea. Parece que arranca bien y avanza unas cuantas yardas, y luego colapsa.*[7]

La construcción de sistemas informáticos es un proceso de entropía decreciente, por lo tanto intrínsecamente metaestable. El mantenimiento del programa es un proceso de entropía creciente, e incluso su más hábil ejecución sólo retrasa el hundimiento del sistema en una irreparable obsolescencia.

12
Herramientas Afiladas

12
Herramientas Afiladas

Un buen artesano se conoce por sus herramientas

PROVERBIO

A. Pisano, “Lo Scultore”, de la Campanile di Santa Maria del Fiore, Florence, c. 1335
Scala/ArtResource, NY

Aun a estas alturas del partido, muchos proyectos de programación todavía operan como talleres mecánicos en lo que a herramientas se refiere. Cada maestro mecánico tiene su propio juego de herramientas personales, coleccionado a lo largo de toda una vida y cuidadosamente guardado bajo llave – la evidencia visible de sus habilidades personales. Del mismo modo, el programador mantiene en reserva en sus archivos pequeños editores, clasificadores, volcados binarios, programas de soporte para el manejo de espacio de disco, etc.

Sin embargo, este enfoque es absurdo en un proyecto de programación. En primer lugar, el problema esencial es la comunicación, y las herramientas individualizadas dificultan en lugar de facilitar la comunicación. En segundo lugar, la tecnología cambia cuando se cambia de computadora o de lenguaje de programación, de tal manera que el ciclo de vida de una herramienta es breve. Por último, obviamente es mucho más eficiente tener un desarrollo y mantenimiento común de herramientas de programación de propósito general.

Sin embargo, las herramientas de propósito general no son suficientes. Tanto las necesidades particulares como las preferencias personales dictan la necesidad por herramientas personalizadas también; en discusiones con equipos de programación he pregonado por un experto en herramientas por equipo. Esta persona deber ser experta en todas las herramientas comunes y debe ser capaz de instruir a su cliente-jefe en su uso. También debe construir herramientas especializadas que su jefe necesita.

Por lo tanto, el gestor de un proyecto debe establecer una filosofía y asignar recursos para la construcción de herramientas comunes. Al mismo tiempo debe reconocer la necesidad por herramientas especializadas, y no escatimar que en sus equipos de trabajo construyan sus propias herramientas. Esta tentación es peligrosa. Uno piensa que si todos estos constructores de herramientas dispersos se reunieran para aumentar el equipo de herramientas comunes, daría como resultado una mayor eficiencia. Pero no es así.

¿Cuáles son las herramientas sobre las que el gestor debe meditar, planificar y organizar? En primer lugar, acerca de una *instalación de cómputo*. Esta requiere computadoras, y se debe adoptar una sistema de horarios de servicio. Necesita un *sistema operativo,* y se deben establecer sistemas de servicio. Requiere un *lenguaje,* y se debe establecer una política del lenguaje. Luego están

los *programas de soporte, – auxiliares de depuración, generadores de casos de prueba* y un *sistema de procesamiento de texto* para el manejo de la documentación. Veamos estos uno por uno.[1]

Computadoras Objetivo

Es útil dividir el apoyo de una computadora en *computadoras objetivo* y *computadoras vehículo*. La computadora objetivo es la única para la cual se escribe el software, y en la que finalmente se harán las pruebas. Las computadoras vehículo son aquellas que proveen los servicios que se usan en la construcción del sistema. Si se está construyendo un nuevo sistema operativo para una computadora antigua, esta puede servir no sólo como una computadora objetivo, sino también como computadora vehículo.

¿Qué tipo de medios objetivo? En la construcción de nuevos programas supervisores u otro software del corazón del sistema los constructores necesitarán, por supuesto, computadoras propias. Tales sistemas necesitarán operadores y un programador de sistemas o dos que mantengan el soporte habitual de la computadora actualizado y en buen estado.

Si se llegara a necesitar otra computadora, lo que es algo bastante raro – no requiere ser rápida, pero debe tener al menos un millón de bytes de almacenamiento principal, un disco en línea de cien millones de bytes y terminales. Solo se requieren terminales alfanuméricas, aunque deben ser más rápidas que los 15 caracteres por segundo que caracterizan a las máquinas de escribir. Una gran memoria aumenta enormemente la productividad ya que permite hacer superposición y ajustar el tamaño del almacenamiento después de las pruebas funcionales.

También se debe adaptar la computadora de depuración, o su software, de tal manera que se puedan realizar automáticamente cuentas y mediciones de todo tipo de parámetros de los programas durante la depuración. Por ejemplo, los patrones de uso de memoria son diagnósticos poderosos de las causas del extraño comportamiento lógico o de la inesperada lentitud en el rendimiento.

Horario Cuando la computadora objetivo es nueva, como cuando se construye su primer sistema operativo, el tiempo de máquina es escaso, y su or-

ganización es un gran problema. La demanda de tiempo de la computadora objetivo tiene una curva peculiar de crecimiento. En el desarrollo del OS/360 teníamos buenos simuladores de la System/360 y otros vehículos. De experiencias previas proyectamos cuántas horas de tiempo de la S/360 necesitaríamos, y empezamos anticipadamente a adquirir computadoras de la fábrica de producción. Pero estuvieron ociosas, mes tras mes. Luego de pronto los 16 sistemas estaban totalmente cargados de trabajo, y entonces el problema ya fue de racionamiento. El uso se veía como la Fig 12.1. Todos empezaron a depurar sus primeros componentes al mismo tiempo, y a partir de entonces gran parte del equipo estaba depurando algo constantemente.

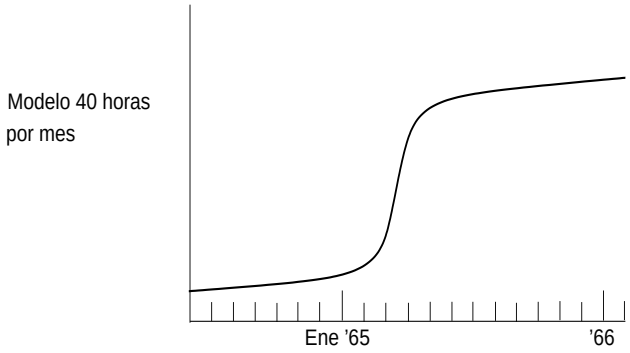


Fig. 11.2 Desarrollo del uso de la computadora objetivo

Centralizamos todas nuestras computadoras y la biblioteca de cintas y establecimos un equipo profesional y experto en centros de cómputo para operarlas. Para maximizar el escaso tiempo de la S/360, realizamos todas las ejecuciones de depuración por lotes en cualquier sistema que estuviera disponible y fuera apropiado. Intentamos cuatro ejecuciones por día (un tiempo de respuesta de dos horas y media) y exigimos un tiempo de respuesta de cuatro horas. Usamos una 1401 auxiliar con terminales para planificar las ejecuciones, mantener el control de los miles de trabajos, y vigilar el tiempo de respuesta.

Pero toda esa estructura resultó estar bastante sobrecargada. Después de unos cuantos meses de un lento tiempo de respuesta, de mutuas recrimina-

ciones, y otras penurias, asignamos tiempo de máquina en bloques significativos. Por ejemplo, a todo el equipo de clasificación de quince miembros se le asignaría un sistema por un bloque de cuatro a seis horas. Esto hizo que ellos mismos se organizaran. Si la computadora estaba ociosa, ningún intruso podría usarla.

Eso, funcionó, fue una mejor forma de asignar y organizar el horario. Aunque el uso de la computadora pudo haber sido un poco menor (y en general no fue así), la productividad estuvo a la alza. Por cada miembro de un equipo de este tipo, diez ejecuciones en un bloque de seis horas son mucho más productivos que 10 ejecuciones en intervalos de tres horas, porque una concentración sostenida reduce el tiempo dedicado a pensar. Después de tal ajetreo, un equipo a menudo necesitaba un día o dos para actualizar la documentación antes de solicitar otro bloque. En general, únicamente tres programadores pueden compartir de forma fructífera un bloque de tiempo. Esta parece ser la mejor forma de administrar una computadora objetivo cuando se depura un nuevo sistema operativo.

En la práctica siempre ha sido así, aunque nunca en la teoría. Depurar un sistema es como la astronomía, ha sido siempre un trabajo nocturno. Hace veinte años, en la 701, se empezó en la productividad informal de las horas previas al amanecer, cuando todos los jefes de los centros de cómputo dormían profundamente en sus hogares, y los operadores no estaban dispuestos a ser estrictos con las reglas. Han pasado tres generaciones de computadoras; la tecnología ha cambiado totalmente; han surgido los sistemas operativos; y aún este método preferido de trabajo no ha cambiado. Ha perdurado porque es más productivo. Ha llegado el momento de reconocer su productividad y adoptar abiertamente esta práctica fructífera.

Computadoras vehículo y Servicios de Datos

Simuladores. Si la computadora objetivo es nueva, se necesita un simulador lógico. Esto brinda un vehículo de depuración mucho antes de la existencia de la verdadera computadora objetivo. E igualmente importante, brinda acceso a un vehículo de depuración *confiable* incluso después de la disponibilidad de una computadora objetivo.

Confiable no es lo mismo que *exacto*. El simulador seguramente fallará en algunos aspectos en ser una implementación fiel y exacta de la arquitectura

de la nueva computadora. Pero será la *misma* implementación día tras día, aunque el nuevo hardware no lo sea.

Hoy en día nos hemos acostumbrado a que el hardware de la computadora funcione correctamente casi todo el tiempo. A menos que un programador de aplicaciones observe que un sistema se comporta de forma inconsistente de una ejecución a otra idéntica, está bastante persuadido a buscar errores en su código en lugar de buscarlos en su hardware.

Sin embargo, esta práctica no es un buen entrenamiento para la programación de apoyo para una nueva computadora. El hardware construido en laboratorio, preproducido, o en su versión temprana *no* trabaja como está definido, *no* funciona de forma confiable, y *no* es estable. A medida que se detectan errores, se hacen cambios de ingeniería en todas las copias de la computadora, incluyendo las del grupo de programación. Esta base móvil es bastante mala. Las fallas de hardware, generalmente intermitentes, son lo peor. La incertidumbre es lo peor de todo, porque roba el incentivo para sumergirse diligentemente en el código buscando algún error – que en todo caso puede no estar allí. Así pues, un simulador confiable en un vehículo bien envejecido conserva su utilidad mucho más de lo que uno esperaría.

Vehículos para compilar y ensamblar. Por las mismas razones, se busca que compiladores y ensambladores se ejecuten en vehículos confiables pero compilen el código objeto para el sistema objetivo. Esto puede entonces empezar a ser depurado en el simulador.

Con la programación en lenguajes de alto nivel, se puede hacer mucho de la depuración compilando y probando el código objeto en la computadora vehículo antes de empezar realmente a probar el código de la computadora objetivo. Esto brinda la eficiencia de la ejecución directa, en lugar de la simulación, combinada con la confiabilidad de una computadora estable.

Bibliotecas de programas y contabilidad. Una aplicación muy exitosa e importante de una computadora vehículo en el desarrollo del OS/360 fue para el mantenimiento de las bibliotecas de programas. Un sistema desarro-llado bajo el liderazgo de W. R. Crowley tenía conectadas dos 7010, que compartían un gran banco de datos en disco. Las 7010 también proporcionaban un ensamblador de la S/360. Todo el código probado o a prueba era almacendado

en esta biblioteca, tanto el código fuente como los módulos de carga ensamblados. De hecho, la biblioteca fue dividida en sub bibliotecas con distintas reglas de acceso.

Primero, cada grupo o programador tenía un área donde mantenía copias de sus programas, sus casos de prueba y el andamiaje necesario para la prueba del componente. En esta área de *corralito* no habían restricciones acerca de lo que un programador podía hacer con sus propios programas; eran suyos.

Cuando una persona tenía un componente listo para la integración dentro de una pieza mayor, entregaba una copia al gestor de ese sistema mayor, quien ponía esta copia dentro de una *sub biblioteca de integración del sistema*. Ahora el programador original no podía cambiarla, excepto con el permiso del gestor de integración. A medidad que el sistema se integraba, este último continuaría con todo tipo de pruebas del sistema, identificando los errores y reparándolos.

De tanto en tanto, una versión del sistema estaría lista para un uso más amplio. Luego, sería promovida a una *sub biblioteca de la versión actual.* Esta copia era sacrosanta, solo se tocaba por errores desvastadores. Estaba disponible para su uso en la integración y pruebas de todas las versiones de los nuevos módulos. Un directorio de programas en la 7010 mantenía un seguimiento de cada versión de cada módulo, su estado, su ubicación y sus cambios.

Aquí son importantes dos ideas. La primera es el *control*, la idea de que las copias del programa pertenecen a los gestores que son los únicos que pueden autorizar sus modificaciones. La segunda es la de la *separación formal* y la *evolución* desde el corralito, a la integración y a la liberación.

En mi opinión esto fue una de las cosas mejor hechas en el trabajo del OS/360. Es una pieza de tecnología de gestión que parece haber sido desarrollada de forma independiente en varios proyectos de programación masivos incluyendo las de Bell Labs, ICL y la Universidad de Cambridge.[8] Es aplicable a la documentación como también a la programación. Es una tecnología indispensable.

Herramientas de programación. A medida que surgen nuevas técnicas de depuración, las antiguas disminuyen pero no desaparecen. Así es como se necesitan volcadores de memoria, editores de código fuente, volcadores de copias del estado del sistema e incluso trazadores.

Asimismo se necesita un conjunto completo de programas de soporte para poner paquetes comprimidos en discos, hacer copias de cintas, imprimir archi-vos, cambiar catálogos. Si se encarga anticipadamente a un fabricante de he-rramientas del proyecto, esto se puede hacer una vez y estar listo cuando sea necesario.

Sistema de documentación. Entre todas las herramientas, la que ahorra la mayor parte del trabajo puede ser un sistema computarizado de edición de texto, operando en un vehículo confiable. Teníamos uno muy práctico, ideado por J. W. Franklin. Sin este sistema sospecho que los manuales del OS/360 hubieran estado bastante atrasados y hubieran sido más crípticos. E-xisten aquellos que argumentarían que el estante de seis pies de los manuales del OS/360 representa una diarrea verbal, que su misma voluminosidad in-troduce un nuevo tipo de incomprensibilidad. Existe algo de cierto en ello.

Pero contestaré de dos manera. Primero, la documentación del OS/360 es abrumadora al por mayor, pero el plan de lectura está diseñado prolija-mente; si uno lo usa de forma selectiva, se puede soslayar la mayor parte del volumen la mayor parte del tiempo. Se debe considerar la documentación del OS/360 como una biblioteca o una enciclopedia, no como un conjunto de textos obligatorios.

Segundo, esto es mucho más preferible a la severa falta de documentación que caracteriza a la mayoría de los sistemas de programación. Sin embargo, sin demora estaré de acuerdo en que la escritura podría mejorarse enorme-mente en ciertos lugares, y que el resultado de una mejor escritura sería un volumen reducido. Algunas partes (e.g., *Conceptos y Medios*) están muy bien escritas ahora.

Simulador de rendimiento. Es mejor tener uno. Constrúyalo desde el punto de vista del usuario, como se discute en el siguiente capítulo. Use el mismo diseño descendente para el simulador de rendimiento, el simulador lógico y el producto. Inícielo muy pronto. Escúchele cuando hable.

Lenguajes de Alto Nivel y Programación Interactiva

Actualmente las dos herramientas más importantes en la programación de sistemas son las que no se usaron en el desarrollo del OS/360 hace casi una

década. Aún no se usan ampliamente, pero todas las pruebas apuntan a su poder y aplicabilidad. Son (1) un lenguaje de alto nivel y (2) la programación interactiva. Estoy convencido de que sólo la inercia y la pereza impiden la adopción universal de estas herramientas; las dificultades técnicas ya no son excusas válidas.

Lenguaje de alto nivel. Las principales razones para usar un lenguaje de alto nivel son la productividad y la velocidad de depuración. Ya discutimos anteriormente la productividad (Capítulo 8). No existe mucha evidencia numérica, pero la que existe sugiere mejoras por factores enteros, no solo porcentajes incrementales.

La mejora en la depuración viene del hecho de que existen menos errores, y son más fáciles de encontrar. Hay menos porque se evita un nivel entero de exposición al error, un nivel en el cual no sólo se cometen errores sintácticos sino también semánticos, como el uso incorrecto de registros. Los errores son más fáciles de encontrar porque los diagnósticos del compilador ayudan a encontrarlos y, más importante aún, porque es muy fácil insertar copias del estado del sistema de depuración.

Para mí, estas razones de la productividad y la depuración son abrumadoras. No puedo concebir fácilmente un sistema de programación construido en lenguaje ensamblador.

Ahora bien, ¿qué hay acerca de las clásicas objeciones a tales herramientas? Son tres: No me permiten hacer lo que quiero. El código objeto es muy grande. El código objeto es muy lento.

Respecto a la funcionalidad, creo que la objeción ya no es válida. Todos los testimonios indican que se puede hacer lo que se necesite hacer, pero que cuesta trabajo descubrir cómo, y uno puede ocasionalmente requerir artificios desagradables.[3,4]

Respecto al espacio, los nuevos compiladores optimizados están empezando a ser muy satisfactorios, y seguirán mejorando.

Respecto a la velocidad, la optimización de los compiladores ahora produce un código que es más rápido que la mayor parte del código escrito a mano por un programador. Además, generalmente se pueden resolver los problemas de velocidad reemplazando del uno al cinco por ciento de un programa generado por el compilador por un sustituto escrito a mano después de haber

depurado el primero totalmente.[5]

¿Qué lenguajes de alto nivel se deberían usar para los sistemas de programación? Actualmente el único candidato razonable es PL/I.[6] Tiene un conjunto muy completo de funciones; es compatible con los ambientes del sistema operativo; y se dispone de una variedad de compiladores, unos interactivos, otros rápidos, unos con mucho diagnóstico, y otros que producen código optimizado de muy alto nivel. Personalmente hallo que es más rápido elaborar algoritmos en APL; luego trasladarlos a PL/I para hacerlos compatibles con el ambiente del sistema.

Programación interactiva. Una de las justificaciones del proyecto Multics del MIT fue su utilidad para construir sistemas de programación. Multics (y le sigue el TSS de IBM) difiere en concepto de otros sistemas de cómputo interactivos precisamente en esos aspectos necesarios para la programación de sistemas: muchos niveles de compartición y protección de datos y programas, una amplia gestión de bibliotecas, y medios para el trabajo cooperativo entre usuarios de terminales. Estoy convencido de que en muchas aplicaciones los sistemas interactivos nunca desplazarán a los sistemas por lotes. Pero pienso que el equipo de Multics ha construido su caso más convincente en la aplicación de la programación de sistemas.

Aún no disponemos de muchas evidencias acerca de la verdadera utilidad de tales herramientas en apariencia poderosas. Hay un amplio consenso acerca de que la depuración es la parte difícil y lenta de la programación de sistemas, además el lento tiempo de respuesta es la ruina de la depuración. Así que la lógica de la programación interactiva parece inexorable.[7]

Programa	Tamaño	Lotes (L) o Conversacional (C)	Instrucciones/hombre-año
código ESS	800,000	L	500-1000
apoyo 7094 ESS	120,000	L	2100-3400
apoyo 360 ESS	32,000	C	8000
apoyo 360 ESS	8,300	L	4000

Fig 12.2 Productividad comparativa en la programación bajo lotes y conversacional

Además, hemos escuchado buenos testimonios de muchos que han construido pequeños sistemas o partes de sistemas de esta manera. Las únicas cifras que he visto para efectos de la programación de sistemas grandes están en el informe de John Harr de Bell Labs. Se muestran en la Fig. 12.2. Estas cifras son para la escritura, el ensamblado y el depurado de programas. El primer programa es principalmente el programa de control; los otros tres son traductores de lenguajes, editores y demás. Los datos de Harr sugieren que un medio interactivo al menos duplica la productividad en la programación de sistemas.[8]

El uso eficaz de la mayoría de las herramientas interactivas requiere que el trabajo se realice en un lenguaje de alto nivel, además para depurar a través del volcado de memoria no podemos usar terminales de teletipo o de máquina de escribir. Con un lenguaje de alto nivel, se facilita la edición del código fuente y también la impresión selectiva. En efecto, juntas conforman un par de herramientas afiladas.

13
El Todo y las Partes

13
El Todo y las Partes

Yo puedo evocar los espíritus del fondo del abismo.
También lo puedo yo y cualquier hombre puede hacerlo; falta saber si vienen, cuando los llamáis.

ENRIQUE IV, PARTE 1, SHAKESPEARE

La magia moderna, como la antigua, tiene sus charlatanes: "Puedo escribir programas que controlen el tráfico aéreo, intercepten misiles balísticos, concilien cuentas bancarias, controlen líneas de producción." A lo cual viene la respuesta, "También yo puedo hacerlo, también lo puede hacer cualquiera, pero ¿funcionarán cuando los escriban?"

¿Cómo se construye un programa para que funcione? ¿Cómo se prueba un programa? ¿Y cómo se integra un conjunto probado de componentes dentro de un sistema probado y confiable? Aquí y allá hemos mencionado acerca de algunas técnicas; ahora las consideraremos de forma algo más sistemática.

Diseño Libre de Errores

A prueba de errores, la definición. Los errores más perniciosos y sutiles son los errores del sistema que surgen de suposiciones erróneas hechas por los autores de distintos componentes. El enfoque de la integridad conceptual discutido arriba en los Capítulos 4, 5, y 6 trata estos problemas directamente. Brevemente, la integridad conceptual del producto no sólo facilita su uso, también facilita su construcción y lo hace menos propenso a errores.

También lo hace el detallado y meticuloso trabajo arquitectónico implícito en dicho enfoque. V. A. Vyssotsky, del Proyecto de Salvaguardia de Bell Telephone Laboratories, dice, "La tarea crucial es conseguir el producto definido. Demasiadas fallas se refieren exactamente a esos aspectos que nunca fueron del todo especificados."[1] Para reducir el número de errores en el sistema se necesita una meticulosa definición de la funcionalidad, una cuidadosa especificación y una disciplinada expulsión de adornos funcionales como de técnicas delirantes.

Prueba de la especificación. Mucho antes de cualquier código en sí, se debe entregar la especificación a un grupo de prueba externo para el escrutinio de su completitud y claridad. Como Vyssotsky dice, los propios desarrolladores no pueden hacer esto: "No te dirán que no lo entienden; y de forma campante inventarán su camino a través de inconsistencias y opacidades."

Diseño descendente. Niklaus Wirth, en un artículo muy claro de 1971 formalizó un procedimiento de diseño que había sido usado por años por los mejores programadores.[2] Además, sus ideas, aunque establecidas para el diseño de programas, se aplican completamente al diseño de sistemas complejos de programas. La división de la construcción de sistemas en arquitectura, implementación y realización es una expresión de estas ideas; además, cada arquitectura, implementación y realización puede ser mejorada por los métodos descendentes.

Brevemente, el procedimiento de Wirth es identificar el diseño como una secuencia de *pasos de refinamiento.* Se esboza una definición aproximada de la tarea y un método de solución aproximado que consiga el resultado principal. Después se examina la definición con mayor atención y se observa cómo difiere el resultado del que se desea, y se desglosan los pasos grandes de la solución en pasos más pequeños. Cada refinamiento en la definición de la tarea llega a ser un refinamiento en el algoritmo de la solución, y cada uno puede ir acompañado por un refinamiento en la representación de datos.

A partir de este proceso, se identifican *módulos* de solución o de datos cuyo refinamiento adicional puede proceder independientemente de otros trabajos. El grado de esta modularidad determina la adaptabilidad y la capacidad de cambio del programa.

Wirth aboga por el uso de una notación de tan alto nivel como sea posible en cada paso, exponiendo los conceptos y ocultando los detalles hasta que llegue a ser necesario un refinamiento adicional.

Un buen diseño descendente evita errores de varias formas. Primero, la claridad de la estructura y su representación facilita el planteamiento preciso de los requisitos y funciones de los módulos. Segundo, el particionamiento e independencia de los módulos evita errores del sistema. Tercero, la supresión de detalles hace que los defectos en la estructura sean más evidentes. Cuarto, el diseño se puede probar en cada uno de sus pasos de refinamiento, así que las pruebas pueden empezar más pronto y concentrarse en el nivel de detalle adecuado en cada paso.

El proceso de refinamiento gradual no significa que jamás tengamos que retroceder o fragmentar el nivel máximo, y empezar todo de nuevo cuando tropecemos con un detalle inesperadamente espinoso. De hecho, eso ocurre con frecuencia. Pero es mucho más fácil ver exactamente cuándo y por qué se

debe descartar un mal diseño y empezar de nuevo. Muchos sistemas pobres provienen de un intento por salvar un mal diseño básico y remendarlo con todo tipo de recursos cosméticos. El diseño descendente reduce esta tentación.

Estoy convencido de que el diseño descendente es la formalización reciente más importante de la década.

Programación estructurada. Otro conjunto importante de nuevas ideas para el diseño de programas libres de errores se deriva en gran parte de Dijkstra,[3] y está construido sobre una estructura teórica realizada por Böhm y Jacopini.[4]

Básicamente el enfoque es diseñar programas cuyas estructuras de control consistan sólo de lazos definidos por una declaracion tal como `DO WHILE`, y secciones condicionales delineadas en grupos de declaraciones marcadas con corchetes y condicionadas por un `IF ... THEN ... ELSE`. Böhm y Jacopini demuestran que estas estructuras son teóricamente suficientes; Dijkstra arguye que la alternativa, la bifurcación irrestricta vía `GO TO`, produce estructuras que se prestan a errores lógicos.

La idea básica es sin duda sólida. Se han hecho muchas críticas y se han elaborado estructuras de control adicionales, tales como una bifurcación de n-caminos (la así llamada declaración `CASE`) para distinguir entre muchas posibilidades, y una salida del desastre (`GO TO ABNORMAL END`) son muy prácticas. Además, algunos se han vuelto muy doctrinarios acerca de evitar todos los `GO TO`'s, eso parece exagerado.

El punto importante y vital para construir programas libres de errores, es que uno quiera pensar acerca de las estructuras de control de un sistema como si fueran estructuras de control, no como declaraciones individuales de bifurcación. Esta forma de pensar es un gran paso adelante.

Depuración de Componentes

Los procedimientos para depurar programas han completado un gran ciclo en los pasados veinte años, y de alguna forma han regresado a donde empezaron. El ciclo ha atravesado por cuatro pasos, y es interesante seguirles la pista y ver las motivaciones de cada uno.

La depuración en computadora. Las primeras computadoras tenían relativamente un pobre equipo de entrada-salida y largas demoras de entrada-salida. Normalmente, la computadora leía y escribía una cinta de papel o una cinta magnética y se usaban medios fuera de línea para la preparación e impresión de la cinta. Esto hizo que la cinta de entrada-salida fuera intolerablemente incómoda para la depuración, así que en su lugar se usó la consola. De esta manera la depuración se diseñó para permitir tantas pruebas como fuera posible por sesión de máquina.

El programador diseñaba meticulosamente su procedimiento de depuración – planificando dónde parar, qué localidades de memoria examinar, qué encontrar ahí y qué hacer si no lo encontraba. Esta programación meticulosa de sí mismo como una máquina de depuración bien podía durar casi la mitad del tiempo dedicado a la escritura del programa a ser depurado.

El pecado cardinal era oprimir `START` audazmente sin haber segmentado el programa en secciones de prueba con paradas planificadas.

Volcados de Memoria. La depuración en computadora fue muy eficaz. En una sesión de dos horas, se podía obtener quizá una docena de ejecuciones. Pero las computadoras eran muy escasas, y muy costosas, y el pensar que todo ese tiempo de máquina se iba a desperdiciar era horripilante.

Así que cuando se conectaron en línea las impresoras de alta velocidad, cambió la técnica. Se ejecutaba un programa hasta que fallaba la verificación, y entonces se volcaba toda la memoria. Luego empezaba la laboriosa tarea de escritorio, verificando el contenido de cada localidad de memoria. El tiempo de escritorio no era muy diferente al tiempo de la depuración en computadora; pero sucedía después de ejecutar la prueba, en el desciframiento, y no como antes, en la planificación. Para cualquier usuario depurar tomaba mucho más tiempo, porque las ejecuciones de prueba dependían del tiempo de respuesta de la ejecución por lotes. Sin embargo, todo el procedimiento fue diseñado para minimizar el tiempo de uso de la computadora, y servir a tantos programadores como fuera posible.

Copias del Estado del Sistema. Las computadoras sobre las que se desarrolló el volcado de memoria tenían de 2000 a 4000 palabras, o de 8K a 16K bytes de memoria. Pero las capacidades de memoria crecían a pasos agigan-

tados, y el volcado total de memoria resultó impráctico. Así que se desarrollaron técnicas de volcado de memoria selectivo, trazado selectivo y para la inserción de copias del estado del sistema dentro de los programas. El OS/360 TESTRAN es el último en esta dirección, permite insertar copias del estado del sistema dentro de un programa sin necesidad de reensamblar o recompilar.

Depuración interactiva. En 1959 Codd, sus colegas[5] y Strachey[8] informaron acerca de su trabajo dirigido a la depuración en tiempo compartido, una manera de lograr, por una lado, una respuesta instantánea de la depuración en computadora y, por otro, el uso eficiente de una computadora de depuración por lotes. La computadora tendría múltiples programas en memoria, listos para su ejecución. Una terminal, controlada solo por programa, estaría asociada con cada programa en proceso de depuración. La depuración estaría bajo control de un programa supervisor. Cuando el programador a través de una terminal detenía su programa para examinar el avance o para hacer cambios, el supervisor ejecutaría otro programa, manteniendo así ocupada la computadora.

El sistema de multiprogramación de Codd fue desarrollado, pero se hizo énfasis en la mejora del rendimiento a través del uso eficiente de la entrada-salida, y no se implementó la depuración interactiva. Las ideas de Strachey fueron mejoradas e implementadas en 1963 en un sistema experimental para la 7090 por Corbató y sus colegas en el MIT.[7] Este desarrollo condujo al MULTICS, TSS, y a otros sistemas de tiempo compartido actuales.

Las principales diferencias percibidas por el usuario entre la depuración en computadora como se practicaba al principio y la depuración interactiva de hoy son los medios hechos posible gracias a la presencia de un programa supervisor y sus intérpretes de lenguajes asociados. Se puede programar y depurar en un lenguaje de alto nivel. Los medios de eficientes de edición facilitan hacer cambios y copias del estado del sistema.

Regresar a la capacidad del tiempo de respuesta instantánea de depuración en computadora todavía no ha llevado de regreso a la planificación de las sesiones de depuración. En cierto sentido tal planificación ya no es tan necesaria como antes, puesto que el tiempo de máquina no se consume mientras uno se sienta y piensa.

Sin embargo, algunos resultados experimentales interesantes de Gold mu-

estran un progreso de tres veces más en la depuración interactiva lograda en la primera interacción de cada sesión como en las interacciones subsecuentes.[8] Esto sugiere poderosamente que no nos estamos dando cuenta del potencial de la interacción debido a la carencia de una sesión planificada. El tiempo ha venido a quitar el polvo a las viejas técnicas en computadora.

Encuentro que el uso apropiado de un buen sistema de terminal requiere dos horas de escritorio por cada dos horas de sesión en la terminal. La mitad de este tiempo se dedica a la limpieza posterior a la última sesión: actualizando la bitácora de depuración, archivando los listados del programa actualizado en mi cuaderno del sistema y explicando fenómenos extraños. La otra mitad se dedica a la preparación: planificando cambios y mejoras y diseñando pruebas detalladas para la próxima sesión. Sin tal planificación, es difícil mantenerse productivo por más de dos horas. Sin la limpieza de la post-sesión, es difícil mantener la serie de sesiones de terminal sistemáticas y avanzando.

Casos de prueba. En cuanto al diseño de procedimientos de depuración y casos de prueba reales, Gruenberger tiene un tratamiento[9] especialmente bueno y existen también tratamientos más cortos en otros textos estándar.[10,11]

La Depuración del Sistema

La parte inesperadamente difícil de la construcción de sistemas de programación es la prueba del sistema. Ya he discutido algunas de las razones tanto de la dificultad como de sus imprevistos. De todo eso, uno debería convencerse de dos cosas: depurar un sistema tomará más tiempo de lo esperado, y su dificultad justifica un enfoque minuciosamente sistemático y planificado. Veamos ahora qué implica este enfoque.[12]

Use componentes depurados. El sentido común, si no una práctica común, dicta que se debe empezar a depurar el sistema sólo después de que las piezas parecen funcionar.

La práctica común se aleja de esto de dos maneras. La primera es el enfoque de atornillar todo y probar. Esto parece estar basado en la idea de que habrán errores del sistema (i.e. interfaz) además de los errores del componente. Cuando más pronto se pongan las piezas juntas, más pronto el error del sistema emergerá. Un enfoque algo menos sofisticado es la idea de usar

las piezas para que se prueben mutuamente, evitando así mucho andamiaje de prueba. Ambos, obviamente, son ciertos, pero la experiencia demuestra que no son toda la verdad – el uso de componentes limpios, depurados ahorra mucho más tiempo en pruebas del sistema que el gastado en el andamiaje y en las exhaustivas pruebas de componentes.

Un poco más sutil es el enfoque del "error documentado". Afirma que un componente está listo para ingresar a la prueba del sistema cuando se han *encontrado* todos los defectos, mucho antes del momento en el que se *corrigieron* todos. Entonces en la prueba del sistema, si la teoría funciona, uno conoce los efectos esperados de estos errores y puede uno ignorar esos efectos, concentrándose en los nuevos fenómenos.

Todo esto es solo una ilusión, inventada para justificar la angustia provocada por el retraso en los calendarios. Uno *no* conoce todos los efectos esperados de errores conocidos. Si las cosas fueran sencillas, las pruebas del sistema no serían difíciles. Además, la reparación de los errores de componentes documentados seguramente inyectará errores desconocidos, y por tanto la prueba del sistema se vuelve confusa.

Construir con abundante andamiaje. Por andamiaje me refiero a todos los programas y datos construidos con fines de depuración que nunca pretendieron formar parte del producto final. Por ello, no es descabellado que en el producto casi la mitad del código lo constituya el andamiaje.

Una forma de andamiaje es el *componente ficticio*, que consta sólo de interfaces y quizás de algunos datos falsos o algunos pequeños casos de prueba. Por ejemplo, un sistema puede incluir un programa de clasificación que no esté todavía terminado. Se pueden probar sus vecinos usando un programa ficticio que únicamente lea y pruebe el formato de entrada de datos, y escupa un conjunto de datos bien formateados sin sentido pero ordenados.

Otra forma es el *archivo miniatura.* Una forma muy común de error del sistema es la malinterpretación de formatos de archivos de cinta y disco. Así que vale la pena construir algunos pequeños archivos que solo tengan unos cuantos registros típicos, pero todas las descripciones, apuntadores, etc.

El caso límite del archivo miniatura es el *archivo ficticio*, el cual realmente no existe en absoluto. El Lenguaje de Control de Tareas del OS/360 proporciona tales medios, y es extremadamente útil para depurar componentes.

Aún otra forma de andamiaje son los *programas auxiliares*. Generadores de datos de prueba, impresiones de análisis especiales y analizadores de tablas de referencias cruzadas, todos ellos son ejemplos de plantillas y accesorios de propósito especial que uno puede desear construir.[13]

Control de cambios. El control riguroso durante la prueba es una de las técnicas impresionantes de la depuración de hardware, y se aplica también a los sistemas de software.

Primero, alguien debe estar a cargo. Él y solo él debe autorizar los cambios en el componente o la sustitución de una versión por otra.

Luego, como ya se discutió, son necesarias copias controladas del sistema: se mantiene una copia bajo llave de las últimas versiones, usada para prueba de componentes; una copia bajo prueba, con las reparaciones instaladas; copias del corralito donde cada persona pueda trabajar aparte en su componente, haciendo correcciones y extensiones.

En los prototipos de la System/360, se observaban ocasionales hebras de cables púrpura entre los cables amarillos normales. Cuando se encontraba un error, se hacían dos cosas. Se ideaba e instalaba en el sistema una reparación rápida, tal que las pruebas pudieran seguir avanzando. Este cambio se exponía con un cable púrpura, tal que sobresalía como un pulgar adolorido. Se ingresaba en la bitácora. Mientras tanto, se preparaba un documento de cambio oficial y se ingresaba en la máquina de automatización del diseño. Finalmente, esto daba como resultado dibujos actualizados y listas de cables y un nuevo panel de respaldo en el que se implementaba el cambio en circuito impreso o cable amarillo. Ahora el modelo físico y el documento coincidían nuevamente, y el cable púrpura dejaba de existir.

La programación necesita una técnica de cable púrpura, y le hace mucha falta un control riguroso y un profundo respeto por el documento que finalmente es el producto. Los ingredientes vitales de tal técnica son el registro de todos lo cambios en un diario y la distinción, llevada de forma destacada en el código fuente, entre parches rápidos y reparaciones pensadas detenidamente, probadas y documentadas.

Añadir un componente a la vez Este precepto, también, es obvio, aunque el optimismo y la pereza nos incitan a violarlo. Hacerlo requiere componentes

ficticios y otro andamiaje, y eso requiere trabajo. Y después de todo, tal vez no sea necesaria toda esa labor? Tal vez no hay errores?

No! Resista la tentación! De eso trata la prueba sistemática del sistema. Se debe suponer que habrán muchos errores y planificar un procedimiento ordenado para eliminarlos.

Note que se deben tener casos de prueba exhaustivos, probando los sistemas parciales cada vez que se agrega una pieza nueva. Y, los componentes antiguos, ejecutados exitosamente en la última suma parcial, deben volver a ejecutarse en la nueva suma parcial para probar la regresión del sistema.

Cuantificar las actualizaciones. A medida que el sistema emerge, los constructores de componentes aparecerán de cuando en cuando, presentando nuevas versiones de sus piezas - más rápidas, más pequeñas, más completas, o supuestamente menos erróneas. El reemplazo de un componente en operación por una versión nueva requiere el mismo procedimiento sistemático de prueba que agregar un nuevo componente, aunque debería tomar menos tiempo, ya que generalmente estarán disponibles casos de prueba más completos y eficientes.

Todo equipo que construye otro componente ha estado usando la versión más recientemente probada del sistema integrado como banco de pruebas para depurar su pieza. Su trabajo se retrasará por tener que cambiar el banco de pruebas subyacente. Obviamente así debe ser. Pero los cambios deben ser cuantificados. Luego, cada usuario tiene un periodo de estabilidad productiva, interrumpido por ráfagas de cambio en el banco de pruebas. Esto parece ser mucho menos perturbador que los constantes vaivenes y sacudidas.

Lehman y Belady ofrecen pruebas de que los cuantos deben ser muy grandes y con intervalos amplios o muy pequeños y frecuentes.[14] De acuerdo a su modelo, la última estrategia está más sujeta a inestabilidad. Mi experiencia lo confirma: En la práctica jamás me arriesgaría con esa estrategia.

Los cambios cuantificados adaptan perfectamente una técnica de cable púrpura. Se mantiene el parche rápido hasta la siguiente liberación normal del componente, que debe incorporar la reparación en forma probada y documentada.

14
Incubando la Catástrofe

14
Incubando la Catástrofe

Nadie ama al portador de malas noticias.

SÓFOCLES

Cómo es que un proyecto se retrasa un año?
.., Un día a la vez.

A. Canova, "Ercole e Lica," 1802. Hércules lanza a su muerte al mensajero Licas, quien inocentemente trajo la prenda de la muerte. Scala/Art Resource, NY

Cuando uno escucha acerca de desastrosos retrasos en el calendario de un proyecto, se imagina que deben haber ocurrido una serie de grandes calamidades. Sin embargo, los desastres generalmente se deben a termitas, no a tornados; y el calendario se ha retrasado imperceptible aunque inexorablemente. De hecho, las mayores calamidades son más fáciles de manejar; se responde con mayor energía, con una reorganización radical, y la creación de nuevos enfoques. Todo el equipo se pone a la altura de las circunstancias.

Pero el retraso diario es más difícil de identificar, más difícil de impedir y más difícil de recuperar. Ayer se enfermó una persona clave y no se pudo realizar la reunión. Hoy todas las máquinas están caídas, porque un relámpago impactó al transformador de potencia del edificio. Mañana las rutinas de disco no iniciarán la prueba, porque el primer disco está retrasado una semana de fábrica. La nieve, las obligaciones como jurado, los problemas familiares, las reuniones de emergencia con clientes, las auditorías ejecutivas – y la lista continua. Cada uno solo pospone alguna actividad por medio o un día. Y el calendario se retrasa, un día a la vez.

Hitos o Piedras de Molino?

Cómo se controla un gran proyecto en un calendario ajustado? El primer paso es *tener* un calendario. Cada uno de los sucesos de la lista, llamados hitos, tiene una fecha. Ya se discutió que la elección de fechas es un problema de estimación y depende de forma crucial de la experiencia.

Para elegir los hitos solo hay una regla pertinente. Los hitos deben ser concretos, específicos, sucesos medibles y estar definidos con el filo de una navaja. Pero tenemos varios contraejemplos, la codificación está "terminada al 90 por ciento" se afirma la mitad del tiempo total dedicado a la codificación. La depuración está "terminada al 99 por ciento" se dice la mayor parte del tiempo. "La planificación está completa" es un suceso que se puede pregonar casi a voluntad.[1]

Por otro lado, los hitos concretos son eventos al 100 por ciento. "Las especificaciones firmadas por arquitectos e implementadores," "el código fuente completo al 100 por ciento, perforado e ingresado a la biblioteca de disco," "la versión depurada pasa todos lo casos de prueba." Estos hitos concretos delimitan las fases vagas de la planificación, la codificación y la depuración.

Es más importante que los hitos estén bien definidos y sin ambigüedades a que sean fácilmente comprobables por el jefe. *Si* el hito está bien definido tal que nadie pueda engañarse, rara vez alguien mentirá acerca del avance del mismo. Pero si el hito es difuso, el jefe generalmente entenderá un informe diferente al que se le entrega. Complementando a Sófocles, nadie disfruta ser portador de malas noticias, de ninguna, así estén suavizadas sin la intención expresa de engañar.

Dos estudios interesantes del comportamiento de la estimación de los contratistas del gobierno en proyectos de desarrollo a gran escala muestran que:

1. Las estimaciones de la duración de una actividad, realizadas y revisadas meticulosamente cada dos semanas antes de comenzar la actividad, no cambian significativamente a medida que se acerca el momento de inicio, no importa cuán incorrectas resulten ser finalmente.
2. *Durante* la actividad, las *sobre*estimaciones de la duración disminuyen constantemente a medida que la actividad avanza.
3. Las *subestimaciones* no cambian significativamente durante la actividad hasta alrededor de tres semanas previas a la conclusión programada.[2]

Los hitos bien definidos son de hecho un servicio para el equipo, y uno que ellos pueden esperar propiamente del gestor. El hito difuso es la carga más pesada con el que se pueda vivir. De hecho es una piedra de molino que tritura la moral, porque es engañoso en cuanto al tiempo perdido hasta que es irremediable. Y un retraso crónico del calendario es un asesino moral.

"De Cualquier Manera, La Otra Pieza Está Retrasada"

El calendario se retrasa un día; y qué? Quién se pone nervioso por un día de retraso? Lo podemos reponer más tarde. De cualquier manera, la pieza en la cual encaja la nuestra está retrasada.

Un entrenador de béisbol reconoce un talento no físico, *el empuje*, como un don imprescindible de los grandes jugadores y los grandes equipos. Es la característica de correr más rápido de lo necesario, de moverse antes de lo necesario, de esforzarse más de lo necesario. También esto es imprescindible

para los grandes equipo de programación. El empuje proporciona la protección, la capacidad de reserva, que permite a un equipo sobrellevar los contratiempos rutinarios, anticipar y prevenir calamidades menores. La respuesta calculada, el esfuerzo medido, son los aguafiestas que inhiben el empuje. Como hemos visto, uno *debe* ponerse nervioso con el retraso de un día. Pues tales son los elementos de la catástrofe.

Pero no todos los retrasos de un día son igualmente desastrosos. Por lo tanto, hace falta un cálculo de la respuesta, aunque se inhiba el empuje. Cómo se sabe qué retraso es importante? No existe substituto para un diagrama PERT o un calendario de camino crítico. Dicha red muestra quién espera qué. Muestra quién está en el camino crítico, donde cualquier retraso mueve la fecha final. También muestra cuánto se puede retrasar una tarea antes de ser movida dentro del camino crítico.

La técnica PERT, estrictamente hablando, es una elaboración de la calendarización del camino crítico en la que se estima tres veces por cada evento, tiempos correspondientes a diferentes probabilidades de cumplir con las fechas estimadas. Dudo que este refinamiento merezca más esfuerzo, aunque por brevedad llamaré a cualquier red de camino crítico un diagrama PERT.

La parte mas valiosa del uso de un diagrama PERT es su elaboración. El diseño de la red, la identificación de las dependencias y la estimación de las etapas todas ellas obligan a tener en cuenta una planificación muy específica muy temprano en un proyecto. El primer diagrama siempre es terrible, y uno inventa y reinventa preparando el segundo diagrama.

A medida que el proyecto avanza, el diagrama PERT proporciona la respuesta a la excusa desmoralizante, “De cualquier manera, la otra pieza está retrasada.” Muestra cuán necesario es el empuje para mantener nuestra pieza fuera del camino crítico, y sugiere formas de recuperar el tiempo perdido en otra parte.

Debajo de la Alfombra

Cuando un gestor de primera línea observa a su pequeño equipo retrasándose, muy rara vez está dispuesto a ir corriendo con el jefe con esta aflicción. El equipo podría ser capaz de arreglarlo, o él debería ser capaz de concebir o reorganizar para solucionar el problema. Entonces por qué preocupar al jefe? Todo bien por el momento. El motivo por el cual el gestor de primera línea

está allí es precisamente para solucionar tales problemas. Además el jefe tiene suficientes preocupaciones reales que exigen su acción como para buscarse otras. Así es que toda la basura se esconde debajo de la alfombra.

Pero todo jefe necesita dos tipos de información, las excepciones al plan que requieren acción y un panorama del estado como formación.[3] Con este fin necesita saber el estado de todos sus equipos. Obtener un panorama veraz del estado no es fácil.

Aquí es donde los intereses del gestor de primera línea y los del jefe tienen un conflicto intrínseco. El gestor de primera línea teme que si informa acerca del problema, el jefe actuará en consecuencia. Entonces su acción tomará la función del gestor, disminuyendo así la autoridad de éste y estropeando sus otros planes. Así que mientras el gestor piense que puede resolverlo solo, no se lo mencionará al jefe.

El jefe dispone de dos técnicas para levantar la alfombra. Se deben usar ambas. La primera es reducir el papel del conflicto y motivar el compartimiento del estado. La otra es tirar de la alfombra.

Reduciendo el papel del conflicto. El jefe debe distinguir primero entre la información de acción y la información de estado. Debe disciplinarse para *no* actuar en los problemas que sus gestores pueden resolver, y *nunca* actuar en los problemas cuando está revisando explícitamente el estado. Una vez supe de un jefe que invariablemente tomaba el teléfono para dar órdenes antes de finalizar el primer párrafo en un informe de estado. Esa respuesta es garantía de acallar por completo la revelación.

Por el contrario, cuando el gestor sabe que su jefe aceptará el informe de estado sin pánico o derecho a apropiación, viene y da una evaluación genuina.

Todo este proceso se facilita si el jefe etiqueta las reuniones, las revisiones y las conferencias como reuniones de *revisión de estado* versus reuniones de *acción sobre el problema*, y se rige él mismo en consecuencia. Obviamente uno puede convocar a una reunión de acción sobre el problema como una consecuencia de la reunión de estado, si cree que un problema está fuera de control. Pero al menos todos sabrán cuál es el marcador y el jefe piensa dos veces antes de atrapar el balón.

Tirando de la alfombra. Sin embargo, es importante tener técnicas de revisión, ya sea en conjunto o no, mediante las cuales se conozca el estado real. El diagrama PERT con sus habituales hitos bien definidos es la base para tal revisión. En un proyecto grande alguien podría querer revisar una parte de él cada semana o darse una vuelta una vez al mes más o menos.

Un documento clave es un informe que muestre los hitos y sus terminaciones reales. La Fig. 14.1 muestra un extracto de dicho informe. Este informe muestra algunos problemas. La aprobación de las especificaciones está retrasada en varios componentes. La aprobación manual (SLR) está retrasada en otro, y uno resulta retrasado al salir del primer estado (Alfa) de la prueba del producto ralizada de forma independiente. Así que tal informe sirve como una agenda para la reunión del 1 de Febrero. Todos saben las preguntas, y el gestor de componentes debería estar preparado para explicar por qué está retrasado, cuándo estará terminado, qué medidas está tomando, y qué ayuda, si es que alguna, necesita del jefe o de grupos colaterales.

V. Vyssotsky de Bell Telephone Laboratories añade la siguiente observación:

> *He encontrado que es práctico llevar tanto fechas "calendarizadas" como "estimadas" en el informe de hitos. Las fechas calendarizadas son propiedad del gestor de proyecto y representan un plan de trabajo coherente para el proyecto como un todo, y que es* a priori *un plan razonable. Las fechas estimadas son propiedad del gestor del nivel más bajo quien tiene conocimiento acerca de la pieza de trabajo en cuestión, y representan su mejor evaluación respecto de cuándo realmente sucederá, dado los recursos que tiene disponibles y de cuando recibió (o tiene compromisos para la entrega de) sus entradas de prerequisitos. El gestor de proyecto tiene que mantener sus manos fuera de las fechas estimadas, y poner el énfasis en la obtención de estimaciones exactas e imparciales en lugar de estimaciones aceptablemente optimistas o las conservadoras autoprotectoras. Una vez que esto queda claramente establecido para todos, el gestor de proyecto puede atisbar muchos escenarios en el futuro donde tendrá problemas si no actúa.*[4]

SYSTEM/360 INFORME DEL RESUMEN DEL ESTADO
OS/360 PROCESADOR DE LENGUAJE + PROGRAMAS DE SERVICIO
ALREDEDOR DE FEBRERO 01.1965

A=APROBADO
C=COMPLETADO

*=REVISADO EN FECHA PLANEADA
NE=NO ESTABLECIDO

PROYECTO	UBICACION	ANUNCIO DE COMPROMISO LIBERACION	OBJETIVO DISPONIBLE APROBADO	SPECS DISPONIBLE APROBADO	SRL DISPONIBLE APROBADO	PRUEBA ALFA ENTRADA SALIDA	PRUEBA COMP INICIO COMPLETO	PRUEBA SYS INICIO COMPLETO	BOLETIN DISPONIBLE APROBADO	PRUEBA BETA ENTRADA SALIDA
SISTEMA OPERATIVO										
12K NIVEL DE DISEÑO (E)										
ENSAMBLADOR	SAN JOSE	04/--/4 C 12/31/5	10/28/4 C	10/13/4 C 01/11/5	11/13/4 C 11/18/4 A	01/15/5 C 02/22/5				09/01/5 11/30/5
FORTRAN	POK	04/--/4 C 12/31/5	10/28/4 C	10/21/4 C 01/22/5	12/17/4 C 12/19/4 A	01/15/5 C 02/22/5				09/01/5 11/30/5
COBOL	ENDICOTT	04/--/4 C 12/31/5	10/28/4 C	10/15/4 C 01/20/5 A	11/17/4 C 12/08/4 A	01/15/5 C 02/22/5				09/01/5 11/30/5
RPG	SAN JOSE	04/--/4 C 12/31/5	10/28/4 C	09/30/4 C 01/05/5 A	12/02/4 C 01/18/5 A	01/15/5 C 02/22/5				09/01/5 11/30/5
UTILERIAS	TIME/LIFE	04/--/4 C 12/31/5	06/24/4 C		11/20/4 C 12/30/4 A					09/01/5 11/30/5
SORT 1	POK	04/--/4 C 12/31/5	10/28/4 C	10/19/4 C 01/11/5	11/12/4 C 11/30/4 A	01/15/5 C 03/22/5				09/01/5 11/30/5
SORT 2	POK	04/--/4 C 06/30/6	10/28/4 C	10/19/4 C 01/11/5	11/12/4 C 11/30/4 A	01/15/5 C 03/22/5				03/01/6 05/30/6
44K NIVEL DE DISEÑO (F)										
ENSAMBLADOR	SAN JOSE	04/--/4 C 12/31/5	10/28/4 C	10/13/4 C 01/11/5	11/13/4 C 11/18/4 A	01/15/5 C 02/22/5				09/01/5 11/30/5
COBOL	TIME/LIFE	04/--/4 C 06/30/6	10/28/4 C	10/15/4 C 01/22/5	12/17/4 C 12/19/4 A	01/15/5 C 02/22/5				09/01/6 05/30/6
NPL	HURSLEY	04/--/4 C 03/31/6	10/28/4 C							
2250L	KINGSTON	03/30/4 C 03/31/6	11/05/4 C	12/08/4 C 01/04/5	01/12/5 C 01/29/5 A	01/15/5 C 01/29/5				01/03/6 NE
2280	KINGSTON	04/--/4 C 09/30/6	11/05/4 C			04/01/5 C 02/30/5				01/28/6 NE
200K NIVEL DE DISEÑO (H)										
ENSAMBLADOR	TIME/LIFE		10/28/4 C							
FORTRAN	POK	04/--/4 C 06/30/6	10/28/4 C	10/16/4 C 01/11/5	11/11/4 C 12/10/4 A	02/15/5 03/22/5				03/01/6 05/30/6
NPL	HURSLEY	04/--/4 C 03/31/7	10/28/4 C			07/--/5				01/--/7
NPL H	POK	04/--/4 C	03/30/4 C			02/01/5 04/01/5				10/15/5 12/15/5

Figura 14.1

La preparación del diagrama PERT es una función del jefe y los gestores que le informan. Su actualización, revisión, y divulgación requiere el escrutinio de un grupo pequeño (de una a tres personas) que sirve como una extensión del jefe. Tal equipo de *Planes y Controles* es invaluable para un gran proyecto. No tiene autoridad excepto para preguntar a toda la línea de gestores cuando tengan que poner o cambiar hitos, y si los hitos se han cumplido. Puesto que el grupo de Planes y Controles maneja todo el papeleo, la carga sobre la línea de gestores se reduce a lo esencial – la toma de decisiones.

Teníamos un grupo de Planes y Controles experimentado, entusiasta y diplomático, dirigido por A. M. Pietrasanta, que dedicó un considerable talento creativo ideando métodos de control eficaces pero no invasivos. Como resultado, encontré que su grupo era ampliamente respetado y más que tolerado. Para un grupo cuyo papel es intrínsecamente el de irritar, esto es mucho más que un cumplido.

Resulta muy redituable la inversión de una modesta cantidad de trabajo calificado en la operación del equipo de Planes y Controles. Esto ayuda mucho más al cumplimiento del proyecto que si estas personas trabajaran directamente en la construcción de programas del producto. Pues el grupo de Planes y Controles es el guardián que visibiliza los retrasos imperceptibles y señala los elementos críticos. Es el sistema de alerta temprana en contra de la pérdida de un año, un día a la vez.

15
La Otra Cara

15
La Otra Cara

Nadie puede poseer lo que no comprende.

GOETHE

Oh, mejor concédanme comentaristas simples,
Que sin profundas investigaciones enturbien el cerebro.

CRABBE

Una reconstrucción de Stonehenge, la mayor computadora no documentada del mundo.
The Bettman Archive

Un programa de computadora es un mensaje de una persona a una máquina. La sintaxis organizada rígidamente y las escrupulosas definiciones existen para lograr que las intenciones sean claras a la torpe máquina.

Pero un programa escrito tiene otra cara, esa que cuenta su historia al usuario humano. Aún para los programas más privados, es necesario cierto tipo de comunicación; al usuario-autor le fallará la memoria, así que necesitará recordar los detalles de su obra.

Cuánto más vital es la documentación de un programa público, cuyo usuario está alejado del autor en el tiempo y el espacio! Para los productos de programación, la otra cara hacia el usuario es tan importante como la cara hacia la máquina.

Muchos de nosotros hemos despellejado en silencio al remoto y anónimo autor de un programa escasamente documentado. Y por eso muchos de nosotros hemos tratado de inculcar en los nuevos programadores una actitud acerca de la programación que los inspire por el resto de su vida, sobreponiéndolos a la pereza y a las presiones del calendario. Con todo hemos fracasado. Creo que hemos usado métodos errados.

Thomas J. Watson, Sr. contó la historia de su primera experiencia como vendedor de cajas registradoras al norte del estado de Nueva York. Entusiasmado, emprendió la marcha con su carreta cargada de máquinas registradoras. Diligentemente recorrió su territorio, pero sin vender ni una sola. Desanimado, informó a su jefe. El jefe de ventas después de escuchar un rato, dijo, “Ayúdame a cargar unas registradoras en la carreta, arrea el caballo, y vamos de nuevo.” Emprendieron la marcha y juntos visitaron un cliente tras otro, con el viejo *enseñando cómo* vender máquinas registradoras. Todo parece indicar que aprendió la lección.

Por varios años he enseñado diligentemente a mi clase de ingeniería de software acerca de lo necesario y apropiado de una buena documentación, exhortándolos cada vez con más fervor y elocuencia. No funcionó. Supuse que habían aprendido cómo documentar correctamente y estaban fallando por la carencia de celo. Entonces intenté cargar algunas cajas registradoras en la carreta, i.e., *enseñándoles* cómo hacer el trabajo. Esto resultó ser mucho más acertado. Así que en lo que resta de este ensayo minimizaré las exhortaciones y me concentraré en el “cómo” de una buena documentación.

Qué Documentación Es Necesaria?

Se requieren diferentes niveles de documentación para el usuario ocasional de un programa, para el usuario que debe depender de un programa, y para el usuario que debe adaptar un programa a cambios en las circunstancias o el propósito.

Para usar el programa. Todo usuario necesita una descripción en prosa del programa. La mayor parte de la documentación fracasa al brindar una muy pequeña visión global. Se describen los árboles, se comentan la corteza y las hojas, pero no existe un mapa del bosque. Para escribir una descripción en prosa útil, retroceda y haga un acercamiento lentamente:

1. *Propósito.* ¿Cuál es la función principal, el motivo del programa?
2. *Ambiente.* ¿En qué máquinas, configuraciones de hardware, y configuraciones de sistemas operativos se ejecutará?
3. *Dominio y rango.* ¿Qué dominio de entrada es válido? ¿Qué rango de salida puede aparecer legítimamente?
4. *Funciones realizadas y algoritmos utilizados.* ¿Qué es lo que hace exactamente?
5. *Formatos de entrada-salida,* precisas y completas.
6. *Instrucciones de operación,* incluyendo el comportamiento final normal y anormal, tal como se observa en la consola y en las salidas.
7. *Opciones.* ¿Qué opciones tiene el usuario acerca de las funciones? ¿Cómo se han especificado esas opciones exactamente?
8. *Tiempo de ejecución.* ¿Qué tiempo toma ejecutar un problema de un tamaño específico sobre una configuración determinada?
9. *Exactitud y verificación.* ¿Cuán precisas se esperan sean las respuestas? Qué medios de verificación de exactitud se han incorporado?

A menudo toda esta información se puede exponer en tres o cuatro páginas. Se requiere poner mucha atención a la concisión y a la precisión. La mayor parte de este documento necesita ser esbozado antes de escribir el programa, pues plasma decisiones básicas de planificación.

Para creerle al programa. La descripción de cómo se usa un programa se debe complementar con información de cómo se sabe que está funcionando. Esto implica casos de prueba.

Cada copia de un programa enviado debería incluir algunos pequeños casos de prueba que puedan ser utilizados rutinariamente para tranquilizar al usuario que tiene una copia fiel y cargada de forma exacta en su máquina.

Luego se necesitan casos de prueba más exhaustivos, que normalmente se ejecutan sólo después de modificar un programa. Estos encajan en tres partes del dominio de los datos de entrada:

1. Casos principales, prueban las principales funciones del programa para datos comúnmente hallados.
2. Casos apenas legítimos, prueban los extremos del dominio de datos de entrada, garantizan que los valores más grandes y más pequeños posibles, y todos los tipos de excepciones funcionan.
3. Casos apenas ilegítimos, prueban las fronteras del dominio desde el otro lado, aseguran que las entradas inválidas provocan mensajes de diagnóstico correctos.

Para modificar un programa. Cuando se adapta o repara un programa se requiere bastante más información. Por supuesto se necesita el detalle completo que va incluido en una lista bien comentada. Para el modificador, como también para el usuario ocasional, hay una necesidad imperiosa por una visión global clara y definida, esta vez de la estructura interna. Cuáles son los componentes de tal visión global?

1. Un diagrama de flujo o una estructura de grafo del subprograma. Se abunda en esto más adelante.
2. Descripciones completas de los algoritmos utilizados u otras referencias de tales descripciones en la literatura.
3. Una explicación acerca de la disposición de todos los archivos usados.
4. Una visión global de la estructura de pase – la secuencia en la cual se traen datos o programas desde cinta o disco – y lo que se consigue en cada pase.
5. Una exposición de las modificaciones consideradas en el diseño original, la naturaleza y ubicación de conexiones y salidas, y divagaciones de las ideas del autor original acerca de qué modificaciones podrían ser deseables y cómo se podría proceder. También son útiles sus observaciones acerca de las trampas escondidas.

La Maldición del Diagrama de Flujo

El diagrama de flujo es una de las piezas más exageradamente sobrevaloradas de la documentación de un programa. Muchos programas no necesitan en absoluto diagramas de flujo; pocos programas necesitan un diagrama de flujo de más de una página.

Los diagramas de flujo muestran la estructura de decisión de un programa, que es sólo un aspecto de su estructura. Muestran la estructura de decisión de una forma bastante elegante cuando el diagrama de flujo cabe en una sola página, pero el resumen fracasa terriblemente cuando tiene múltiples páginas, remendado con salidas numeradas y conectores.

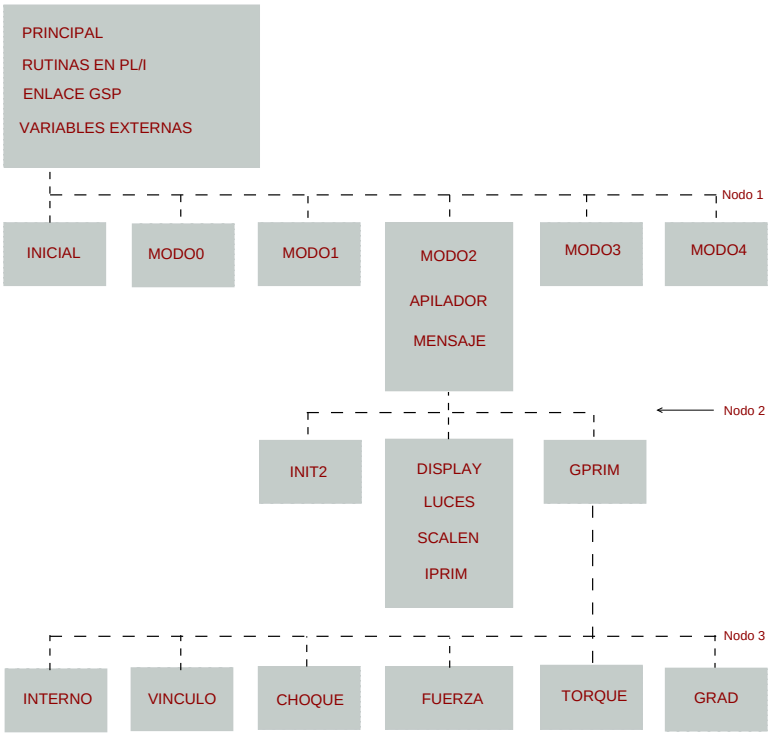


Fig. 15.1 Estructura de grafo de un programa. (Cortesía de W.V. Wright)

El diagrama de flujo de una página para un programa considerable viene a ser esencialmente un diagrama de la estructura del programa, y de las fases o pasos. Como tal es muy práctico. La Figura 15.1 muestra tal estructura de grafo de un subprograma.

Por supuesto dicha estructura de grafo ni sigue ni necesita los estándares de diagramas de flujo de ANSI forjados con esmero. Todas las reglas acerca de formas de caja, conectores, números, etc. son necesarias sólo para dar inteligibilidad a los diagramas de flujo detallados.

Sin embargo, el diagrama de flujo minuciosamente detallado es una molestia obsoleta, adecuado sólo para iniciar a los principiantes en el pensamiento algorítmico. Cuando fue introducido por Goldstine y von Neumann,[1] las pequeñas cajas y sus contenidos servían como un lenguaje de alto nivel, agrupaban las inescrutables declaraciones del lenguaje de máquina en grupos significativos. Como reconoció pronto Iverson,[2] en un lenguaje de alto nivel sistemático la agrupación ya está hecha, y cada caja contiene una declaración (Fig. 15.2). Entonces, las cajas mismas vienen a ser nada más que un ejercicio de dibujo tedioso y devorador de espacio que también podrían ser eliminadas. Finalmente solo nos quedan las flechas. Las que unen una declaración con su sucesor son redundantes; bórrelas también. Eso solo deja `GO TO`'s. Y si se sigue una buena práctica y se usa la estructura de bloque para minimizar los `GO TO`'s, ya no habrán muchas flechas, aunque ayudan enormemente a la comprensión. También se los podría dibujar en el listado y eliminar el diagrama de flujo del todo.

De hecho, los diagramas de flujo se pregonan más de lo que se practican. Nunca he visto a un experimentado programador que rutinariamente haga detallados diagramas de flujo antes de empezar a escribir programas. En lugares donde los estándares de la organización requieren diagramas de flujo, se realizan casi invariablemente después del hecho. Muchas tiendas orgullosamente usan programas para generar esta "indispensable herramienta de diseño" a partir del código final. Creo que este ejercicio generalizado no es una desviación vergonzosa y deplorable de la buena práctica, que deba ser aceptada sólo con una risa nerviosa. En cambio, es la aplicación del buen juicio, y nos enseña algo acerca de la utilidad de los diagramas de flujo.

El apóstol Pedro dijo de los nuevos Gentiles convertidos y la ley Judía, "¿Por qué llevar una carga sobre sus espaldas que ni nuestros ancestros ni

nosotros mismos fuimos capaces de soportar?" (Actos 15:10 TEV[†]). Me gustaría decir lo mismo acerca de los nuevos programadores y la obsoleta práctica de los diagramas de flujo.

Programas Auto-Documentados

Un principio básico del procesamiento de datos nos muestra la insensatez de tratar de mantener archivos independientes sincronizados. Es mucho mejor combinarlos en un solo archivo donde cada registro incorpora toda la información de ambos archivos y lleva el control de acuerdo a una determinada clave.

Todavía, nuestra manera de documentar programas viola nuestras propias enseñanzas. En general tratamos de mantener el documento de un programa legible por la máquina y un conjunto independiente de documentos legible por el humano, que consta de prosa y diagramas de flujo.

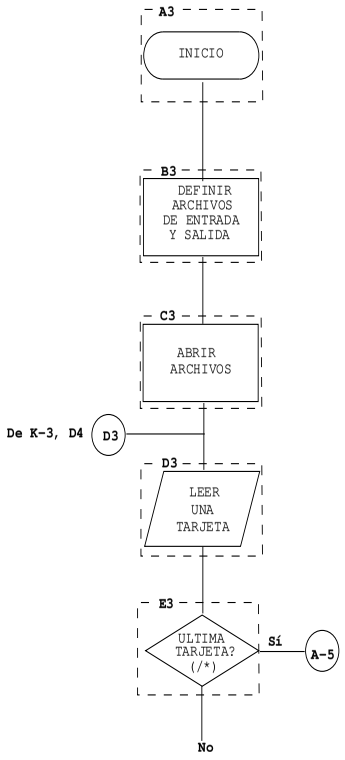
Los resultados de hecho confirman nuestras enseñanzas acerca de la insensatez de mantener archivos separados. La documentación del programa es claramente pobre y su mantenimiento peor. Los cambios realizados al programa no siempre aparecen ni en seguida, ni exactamente en el documento.

Creo que la solución es mezclar los archivos, e incorporar la documentación en el programa fuente. Esto a su vez es un fuerte incentivo hacia el mantenimiento apropiado, y nos garantiza de que la documentación siempre estará a disposición del usuario del programa. Tales programas se llaman *auto-documentados.*

Ahora, si se incluyeran los diagramas de flujo esto claramente sería incómodo (pero no imposible). Pero si admitimos la obsolescencia de los diagramas de flujo y el uso dominante de los lenguajes de alto nivel, resulta razonable combinar el programa y la documentación.

El uso de un programa fuente como un medio de documentación impone ciertas restricciones. Por otro lado, la estrecha disponibilidad que tiene el lector de la documentación del programa fuente, línea por línea, abre la posibilidad a nuevas técnicas. Ha llegado el momento de idear nuevos enfoques y métodos radicales para la documentación de programas.

[†]TEV-Today English Version - Versión en Inglés actual. N.T.




```
PGM4:  PROCEDURE  OPTIONS (MAIN):



           DECLARE SALEFL FILE
           RECORD
           INPUT
           ENVIRONMENT (F(80) MEDIUM (SYSIPT,2501));
     DECLARE PRINT4 FILE
           RECORD
           OUTPUT
           ENVIRONMENT (F(132) MEDIUM (SYSLST,1403) CTLASA);
     DECLARE 01 SALESCARD
           03 BLANK1          CHARACTER (9),
           03 SALESNUM        PICTURE '9999',
           03 NAME            CHARACTER (25),
           03 BLANK2          CHARACTER (7),
           03 CURRENT_SALES   PICTURE '9999V99',
           03 BLANK3          CHARACTER (29);
     DECLARE 01 SALESLIST,
           03 CONTROL         CHARACTER(1) INITIAL (' '),
           03 SALESNUM_OUT    PICTURE 'ZZZ9',
           03 FILLER1         CHARACTER (5) INITIAL (' '),
           03 NAME_OUT        CHARACTER (25),
           03 FILLER2         CHARACTER(5) INITIAL (' '),
           03 CURRENT_OUT     PICTURE 'Z,ZZZV.99',
           03 FILLER3         CHARACTER (5) INITIAL (' '),
           03 PERCENT         PICTURE 'Z9',
           03 SIGN            CHARACTER (1) INITIAL ('%'),
           03 FILLER4         CHARACTER (5) INITIAL (' '),
           03 COMMISSION      PICTURE 'Z,ZZV.99',
           03 FILLER5         CHARACTER (63) INITIAL (' ');


      OEN FILE (SALEFL),FILE (PRINT4);


           ON ENDFILE (SALEFL) GO TO ENDOFJOB;
```

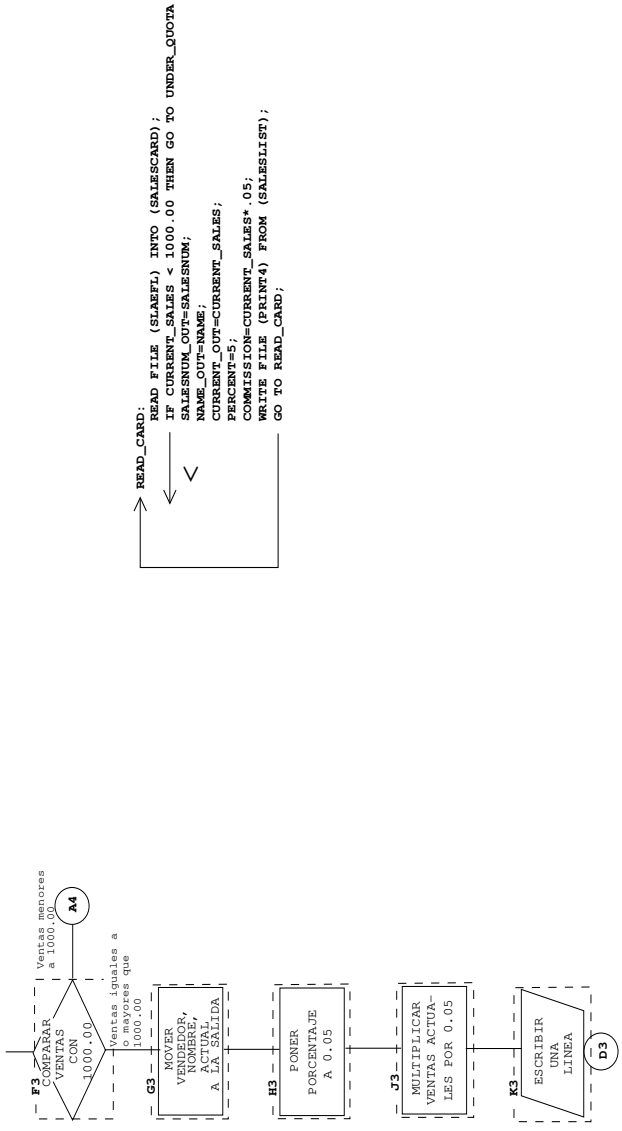


Fig. 15.2 Comparación de un diagrama de flujo y un programa correspondiente en PL/I. [Compendiado y adaptado de las Figs. 15-41, 15-44, del *Data Processing and Computer Programming: A Modular Approach* de Thomas J. Cashman y William J. Keys (Harper & Row, 1971).]

El principal objetivo es minimizar la carga de la documentación, esa carga que nuestros predecesores ni nosotros hemos sido capaces de llevar con éxito.

Una aproximación. Lo primero es usar las partes del programa que tienen que estar de cualquier manera, por razones del lenguaje de programación y para apoyar tanto como sea posible la documentación. Así que aprovechamos etiquetas, declaraciones y nombres simbólicos para la tarea de transmitir al lector tanto significado como sea posible.

Lo segundo es usar el espacio y el formato tanto como sea posible para mejorar la legibilidad y mostrar la subordinación y el anidamiento.

Lo tercero es insertar la documentación en prosa necesaria dentro del programa como párrafos de comentarios. La mayoría de los programas tienden a tener bastantes comentarios línea por línea, esos programas producidos para cumplir rígidos estándares organizativos para "una buena documentación" con frecuencia exageran. Sin embargo, incluso estos programas son generalmente deficientes en los párrafos de comentarios que realmente dan inteligibilidad y una visión global.

Puesto que la documentación está integrada dentro de la estructura, identificaciones y formatos del programa, gran parte de ella *debe* hacerse cuando el programa se escribe por primera vez. Y es cuando *debería* escribirse. Puesto que el enfoque de la auto-documentación minimiza el trabajo extra, por lo tanto existen menos obstáculos para hacerlo.

Algunas técnicas. La Figura 15.3 muestra un programa[3] en PL/I auto-documentado. Los números en círculos no son parte de él; son meta-documentación con clave para la discusión.

1. Use un nombre de trabajo separado por cada ejecución, y mantenga una bitácora de ejecuciones que muestre qué se intentó, cuándo y los resultados. Si el nombre está compuesto de una parte mnemónica (aquí *QLT*) y un sufijo numérico (aquí *4*), el sufijo se puede usar como número de ejecución, vinculando así listados y bitácora. Esta técnica requiere una tarjeta de nuevo trabajo por cada ejecución, aunque puede integrarse en lotes, duplicando la información común.

```
(1) //QLT4 JOB ...

(2) QLTSRT7: PROCEDURE (V):


    /*****************************************************************************************/
(3) /*SUBRUTINA DE CLASIFICACION PARA 2500 CAMPOS DE 6-BYTES, PASADOS COMO EL VECTOR V. UN    */
    /*PROCEDIMIENTO, NO-PRINCIPAL, COMPILADO POR SEPARADO, EL CUAL DEBE USAR UNA ASIGNACION   */
    /*AUTOMATICA DEL NUCLEO.                                                                  */
    /*                                                                                        */
(4) /*EL ALGORITMO DE CLASIFICACION SIGUE EL TEXTO AUTOMATIC DATA PROCESSING DE BROOKS E IVERSON,*/
    /*PROGRAMA 7.23, P. 350. ESE ALGORITMO FUE REVISADO COMO SIGUE:                           */
(5) /*  LOS PASOS 2-12 ESTAN SIMPLIFICADOS PARA M=2.                                          */
    /*  EL PASO 18 ESTA EXPANDIDO PARA MANEJAR EL INDEXADO EXPLICITO  DEL VECTOR DE SALIDA.   */
    /*  SE USO TODO EL CAMPO COMO LA LLAVE DE CLASIFICACION.                                  */
    /*  EL MENOS INFINITO ESTA REPRESENTADO POR CEROS.                                        */
    /*  EL MAS INFINITO ESTA REPRESENTADO POR UNOS.                                           */
    /*  LOS NUMEROS DE LAS DECLARACIONES EN EL PROG.7.23 ESTAN REFLEJADOS EN LAS ETIQUETAS DE LAS*/
    /*     DECLARACIONES DE ESTE PROGRAMA.                                                    */
    /*  UNA CONSTRUCCION IF-THEN-ELSE REQUIERE REPETICIONES DE UNAS CUANTAS LINEAS.           */
    /*                                                                                        */
    /*PARA CAMBIAR LA DIMENSION DEL VECTOR A SER CLASIFICADO, SIEMPRE CAMBIE LA INICIALIZACION */
    /*DE T. SI EL TAMAÑO EXCEDE LOS 4096, CAMBIE TAMBIEN EL TAMAÑO DE T.                      */
    /*UNA VERSION MAS GENERAL PARAMETRIZARIA LA DIMENSION DE V.                               */
    /*                                                                                        */
    /*EL VECTOR DE ENTRADA QUE SE PASA ES REEMPLAZADO POR EL VECTOR DE SALIDA REORDENADO.     */
    /*****************************************************************************************/

(6) /* LEYENDA   (INDEXANDO ORIGEN-CERO)                                                     */


   DECLARE
     (H,                          /*INDICE PARA INICIALIZAR T                                */
      I,                          /*INDICE DEL CAMPO A SER REEMPLAZADO                       */
      J,                          /*INDICE INICIAL DE LAS BIFURCACIONES DESDE EL NODO I      */
      K)  BINARY FIXED,           /*INDICE EN VECTOR DE SALIDA                               */

     (MINF,                       /*MENOS INFINITO                                           */
      PINF)  BIT (48),            /*MAS INFINITO                                             */

      V (*)    BIT (*),           /*SE PASA EL VECTOR PARA SER CLASIFICADO Y REGRESADO        */

      T (0:8190)  BIT (48);       /*ESPACIO DE TRABAJO QUE CONSISTE DEL VECTOR A SER CLASIFICADO, LLENADO*/
                                  /*SALIDA CON INFINITOS, PRECEDIDO POR NIVELES INFERIORES    */
                                  /*LLENAR CON MENOS INFINITOS                    */


   /* AHORA INICIALIZAR PARA LLENAR LOS NIVELES FICTICIOS, EL NIVEL MAXIMO, Y UNA PARTE NO USADA DEL*/
   /* NIVEL SUPERIOR SEGUN SEA NECESARIO.                                                    */


(7) INIT: MINF= (48)  '0'B;
          PINF= (48)  '1'B;


           DO L=   0 TO 4094;   T(L) = MINF;         END;
           DO L=   0 TO 2499;   T(L+4095) = V(L);    END;
           DO L=6595 TO 8190;   T(L) = PINF;         END;


(8) K0:   K = -1;                                                                    (10)
    K1:   I = 0;                  /*         (11)                                  <------| */
    K3:   J = 2*I+1;              /*PONE J A EXPLORAR BIFURCACIONES DESDE EL NODO I.  <-----|| */
    K7:   IF T(J) <= T(J+1)       /*ESCOGE LA BIFURCACION MAS PEQUEÑA.                  >  || */
           THEN                   /*                                                  |||| */
          (9) DO;   (12)          /*                                                  ||| */
    K11:        T(I) = T(J);      /*REEMPLAZA                                         ||| */
    K13:        IF T(I) = PINF THEN GO TO K16; /*SI ES INFINITO, REEMPLACE          +∞  ||| */
                                  /* ESTA TERMINADO                                  |||| */
    K12:        I = J;            /*PONE EL INDICE A UN NIVEL MAS ALTO                |||| */
              END;                /*                                                   |||| */
           ELSE                   /*                                                <---+|| */
             DO;                  /*                                                   | || */
    K11A:       T(I) = T(J+1);    /*                                                   | || */
    K13A:       IF T(I) = PINF THEN GO TO K16;          /*                          +∞ | || */
    K12A:       I = J+1;        /*                                                    | || */
              END;                /*                                                 < | || */
    K14:  IF 2*I < 8191 THEN GO TO K3;    /*REGRESA SI NO ESTA EN EL NIVEL SUPERIOR  ----+|| */
    K15:  T(I) = PINF;          /*SI ES EL NIVEL SUPERIOR, LLENAR CON INFINITO        |  | */
    K16:  IF T(0) = PINF THEN RETURN;     /*PRUEBA EL FIN DE LA CLASIFICACION      <---|  | */
    K17:  IF T(0) = MINF THEN GO TO K1;   /*LIMPIA LOS FICTICIOS INICIALES          −∞     | */
    K18: K = K+1;                 /*INDICE DE PASO DE ALMACENAMIENTO                     | */
         V(K) = T(0); GO TO K1; (12)      /*ALMACENA EL CAMPO DE SALIDA                -------| */
```

```
END QLSRT7;
```

Fig. 15.3 Un programa auto-documentado

2. Use un nombre de programa que sea un mnemónico pero también que contenga un identificador de la versión. Eso es, suponer que habrán varias versiones. El índice aquí es el dígito de menor orden del año 1967.
3. Incorpore la descripción en prosa como comentarios al `PROCEDURE`.
4. Haga referencia a la literatura estándar para documentar los algoritmos básicos donde sea posible. Esto ahorra espacio, generalmente apunta a un tratamiento más completo de lo que uno proporcionaría, y permite al lector versado omitirlo con la confianza de que lo ha comprendido.
5. Muestre la relación con el libro de algoritmos:
   a) cambios b) especialización c) representación
6. Declare todas las variables. Use mnemónicos. Use comentarios para convertir `DECLARE` en una referencia completa. Note que ya contiene nombres y descripciones estructurales, sólo se necesita agregar descripciones de *propósito*. Haciéndolo aquí de esa manera, se puede evitar repetir nombres y descripciones estructurales en un tratamiento separado.
7. Marque la inicialización con una etiqueta.
8. Etiquete las declaraciones en grupos para mostrar las correspondencias con las declaraciones en la descripción del algoritmo en la literatura.
9. Use sangría para mostrar la estructura y el agrupamiento.
10. Añada flechas del flujo lógico a mano en los listados. Son muy útiles en la depuración y cambios. Pueden incorporarse en el margen derecho del espacio de comentarios, haciéndolos parte del texto legible por la máquina.
11. Use líneas de comentarios o indique todo lo que no sea obvio. Si se usaron las técnicas de arriba, serán breves y menores en número de lo que se acostumbra.
12. Ponga múltiples declaraciones en una línea, o una declaración en varias líneas para encajar agrupaciones de pensamiento y mostrar la correspondencia con otras descripciones algorítmicas.

Por qué no? ¿Cuáles son las desventajas de este enfoque de documentación? Hay varias, algunas han sido reales pero se están volviendo imaginarias con el paso del tiempo.

La objeción más seria es el incremento del tamaño del código fuente que debe almacenarse. A medida que la disciplina se mueve cada vez más hacia el almacenamiento del código fuente en línea, esto ha llegado a tener mayor consideración. Yo mismo me hallé siendo más breve en los comentarios a un programa en APL, el cual residirá en disco, que en uno en PL/I que almacenaré como tarjetas.

Simultáneamente todavía nos estamos moviendo hacia el almacenamiento en línea de los documentos en prosa para el acceso y la actualización vía un editor de texto. Como se mostró arriba, amalgamando prosa y programa se *reduce* el número total de caracteres a ser almacenado.

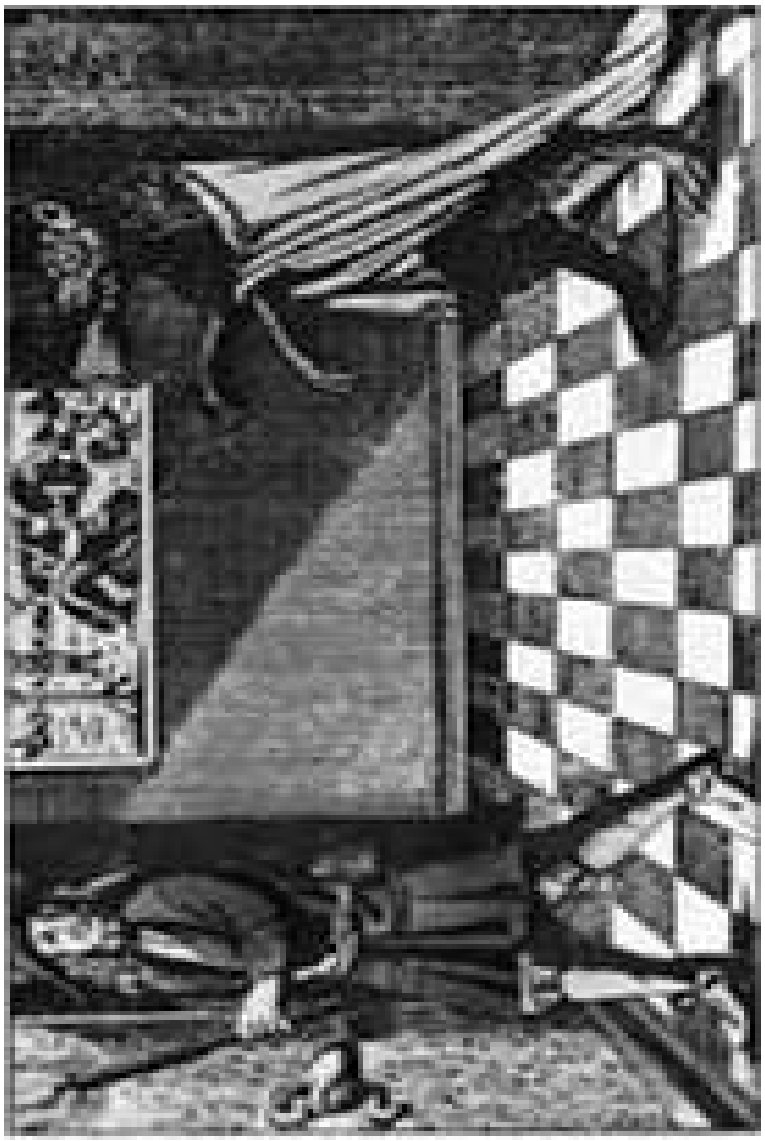
Una respuesta similar se aplica a los argumentos de que los programas auto-documentados requieren más golpes de tecla. La edición de un documento requiere al menos un golpe de tecla por caracter y por documento. Un programa auto-documentado tiene menos caracteres en total y también menos golpes de tecla por caracter, puesto que los documentos no se editan nuevamente.

¿Qué hay de los diagramas de flujo y las estructuras de grafos? Si utilizamos una sola estructura de grafos del más alto nivel, se podría mantener de forma segura como un documento separado, porque no está sujeto a cambios frecuentes. Aunque puede ser incorporado, desde luego, dentro del programa fuente como comentario, esto parece ser prudente.

¿En qué medida las técnicas utilizadas más arriba son aplicables a los programas en lenguaje ensamblador? Pienso que el enfoque fundamental de la auto-documentación es perfectamente aplicable. El espacio y los formatos son menos libres, y así no los podemos usar con tanta flexibilidad. Seguramente se pueden aprovechar nombres y declaraciones estructurales. Los macros pueden ser una gran ayuda. El amplio uso de párrafos de comentarios es una buena práctica en cualquier lenguaje.

Aunque se ha estimulado el enfoque de la auto-documentación a través del uso de lenguajes de alto nivel y ha encontrado su mayor aporte y justificación en los lenguajes de alto nivel usados con sistemas en línea, ya sea por lotes o interactivo. Como he sostenido, tales lenguajes y sistemas ayudan a los programadores en formas muy eficaces. Puesto que las máquinas están hechas para las personas y no las personas para las máquinas, su uso cobra sentido de muchas formas, tanto económicas como humanas.

16
No Existen Balas de Plata - Lo Esencial y lo Accidental en la Ingeniería de Software



16
No Existen Balas de Plata - Lo Esencial y lo Accidental en la Ingeniería de Software

No existe un solo desarrollo, ya sea tecnológico o de gestión, que por sí solo prometa, en una década, una mejora de al menos un orden de magnitud en la productividad, confiabilidad o simplicidad.

El hombre lobo de Eschenbach, Alemania: grabado de línea, 1685. Cortesía de The Grainger Collection, Nueva York.

Resumen[1]

Toda construcción de software involucra tareas esenciales, es decir, la creación de estructuras conceptuales complejas que forman las entidades abstractas de software. Y, tareas accidentales, esto es, la representación de estas entidades abstractas en lenguajes de programación y su mapeo en lenguajes de máquina sujetos a restricciones de espacio y velocidad. En el pasado, la mayoría de los grandes incrementos en la productividad del software han venido de la eliminación de las barreras artificiales que las tareas accidentales extremadamente difíciles ocasionaban, tales como las severas restricciones del hardware, los lenguajes de programación torpes y la falta de tiempo de máquina. ¿Cuánto del quehacer de los ingenieros de software actualmente está todavía dedicado a lo accidental, como opuesto a lo esencial? A menos que sea más de 9/10 del esfuerzo total y reduciendo el tiempo de todas las tareas accidentales a cero no dará un aumento de un orden de magnitud.

Por lo tanto, parece que ha llegado el momento de abordar las partes esenciales de la labor del software, aquellas concernientes con la creación de estructuras conceptuales abstractas de gran complejidad. Sugiero:

- Explotar el mercado masivo para evitar construir lo que se pueda comprar.
- Usar prototipos rápidos como parte de una iteración planificada para establecer los requisitos del software.
- Desarrollar el software orgánicamente, añadiendo más y más funciones a los sistemas a medida que se ejecuten, usen y prueben.
- Identificar y desarrollar a los grandes diseñadores conceptuales de la nueva generación.

Introducción

De todos los monstruos que pueblan las pesadillas de nuestro folklore, ninguno aterra más que los hombres lobo, pues se transforman inesperadamente de algo familiar en horror. Es por ellos que buscamos balas de plata que mágicamente los puedan sepultar.

Los típicos proyectos de software comparten algunas de estas característica (al menos tal como lo ve el gestor no técnico), generalmente ingenuo y

sincero, pero capaz de convertirse en el monstruo de los plazos no cumplidos, de los presupuestos rebasados y productos defectuosos. Por eso escuchamos lamentos desesperados por una bala de plata, algo que haga disminuir los costos del software tan rápidamente como lo hacen los costos del hardware.

Pero como se ve el horizonte de aquí a una década, no se observan balas de plata. No existe un solo desarrollo, ni en la tecnología ni en la gestión, que por sí solo prometa al menos una mejora de un orden de magnitud en la productividad, confiabilidad o simplicidad. En este capítulo trataremos de ver por qué, examinaremos tanto la naturaleza del problema del software como las propiedades de las balas que han sido propuestas.

Sin embargo, no es lo mismo ser escéptico que pesimista. A pesar de que no se vislumbran avances sorprendentes, y de hecho, creo que es inconsistente con la naturaleza del software, están en marcha muchas innovaciones promisorias. Un esfuerzo disciplinado y consistente para desarrollar, propagar y explotar estas innovaciones debería en efecto producir una mejora de un orden de magnitud. No hay un camino sin obstáculos, pero hay uno.

El primer paso hacia la cura de la enfermedad fue la sustitución de teorías demoníacas y de humores por la teoría del germen. Solo ese paso en sí, el comienzo de la esperanza, se deshizo de todas las esperanzas en soluciones mágicas. Esto mostró a los programadores que el progreso sería realizado paso a paso, con gran esfuerzo, y que se tendría que prestar una atención persistente e incesante al hábito de la pulcritud. Así se encuentra hoy la ingeniería de software.

Tiene Que Ser Difícil? – Dificultades Esenciales

No sólo no existen balas de plata a la vista, la propia naturaleza del software hace improbable que exista una – ninguna invención hará por la productividad, la confiabilidad y la simplicidad del software lo que la electrónica, los transistores y la integración a gran escala hicieron por el hardware de computadoras. No podemos esperar ver incrementos que se dupliquen cada dos años.

En principio, debemos observar que la anomalía no es que el progreso del software sea tan lento sino que el progreso del hardware sea tan rápido. Ninguna otra tecnología desde el inicio de la civilización ha visto incrementos en la relación precio-rendimiento de seis órdenes de magnitud en 30 años.

En ninguna otra tecnología se puede escoger en obtener beneficios ya sea aumentando el rendimiento o reduciendo costos. Estos beneficios provienen de la transformación de la manufactura de computadoras a partir de una industria de ensamblaje a una industria de procesos.

En segundo término, para ver qué tasa de progreso podemos esperar en la tecnología del software, examinemos sus dificultades. De acuerdo con Aristóteles, las dividimos en *esencia* – las dificultades intrínsecas a la naturaleza del software – y *accidentes* – aquellas dificultades que hoy enfrentamos en su producción pero que no son intrínsecas.

Los accidentes los discutimos en la próxima sección. Consideremos primero la esencia.

La esencia de una entidad de software es una construcción de conceptos interrelacionados: conjuntos de datos, relaciones entre campos de datos, algoritmos e invocaciones de funciones. Esta esencia es abstracta, en el sentido que la construcción conceptual es la misma independientemente de su representación. No obstante, es muy precisa y rica en detalles.

Creo que la parte más difícil de la construcción del software es la especificación, el diseño y las pruebas de esta construcción conceptual, no el trabajo de representarlo y probar la fidelidad de dicha representación. Y, con seguridad, todavía cometemos errores de sintaxis; pero son solo detalles comparados con los errores conceptuales en la mayor parte de los sistemas.

Si esto es cierto, la construcción del software siempre será difícil. Por su propia natualeza no existen balas de plata.

Consideremos las propiedades intrínsecas de esta esencia irreductible de los sistemas modernos de software: la complejidad, conformidad, variabilidad e invisibilidad.

Complejidad. Las entidades de software son más complejas por su tamaño que tal vez cualquier otra construcción humana, porque no hay dos partes iguales (al menos encima del nivel declarativo). Si existen, hacemos que las dos partes similares sean una sola subrutina, abierta o cerrada. En este sentido los sistemas de software difieren profundamente de las computadoras, los edificios o automóviles, donde la repetición de elementos existen en abundancia.

Las computadoras digitales son en sí más complejas que la mayoría de las

cosas que se construyen; tienen un número muy grande de estados. Esto hace que sea difícil concebirlos, describirlos y probarlos. Los sistemas de software tienen más estados de órdenes de magnitud que las computadoras.

Del mismo modo, hacer una entidad de software a mayor escala no es únicamente una repetición de los mismos elementos en grande; necesariamente es un incremento en el número de elementos diferentes. En la mayoría de los casos, los elementos interactúan entre sí de una forma no lineal, y la complejidad del todo se incrementa mucho más que linealmente.

La complejidad del software es una propiedad esencial, no accidental. Por lo tanto, las descripciones de una entidad de software que omiten su complejidad muchas veces también omiten su esencia. Durante tres siglos, las matemáticas y las ciencias físicas dieron grandes pasos en la construcción de modelos simplificados de fenómenos complejos, derivando propiedades de estos modelos y verificándolas experimentalmente. Esto funcionó porque las complejidades ignoradas en los modelos no eran propiedades esenciales del fenómeno. Esto no funciona cuando las complejidades son la esencia.

Muchos de los problemas clásicos del desarrollo de productos de software se derivan de su complejidad esencial y su incremento no lineal con el tamaño. A partir de la complejidad deriva la dificultad de comunicación entre los miembros del equipo, lo cual conduce a defectos del producto, costos excesivos y retrasos en el calendario. De la complejidad viene la dificultad de enumerar, ni hablar de comprender, todos los posibles estados del programa, y de ahí viene la falta de confiabilidad. De la complejidad de las funcionalidades procede la dificultad de invocarlas, lo cual hace que los programas sean difíciles de usar. De la complejidad de la estructura viene la dificultad de la extensión del programa hacia nuevas funcionalidades sin crear efectos laterales. De la complejidad de la estructura provienen los estados no visualizados que constituyen agujeros de seguridad.

De la complejidad no sólo se derivan problemas técnicos, sino también problemas de gestión. Esta complejidad dificulta la visión global, impidiendo así la integridad conceptual. Dificulta encontrar y controlar todos los cabos sueltos. Crea una tremenda carga de aprendizaje y compresión que hace que la rotación del personal sea un desastre.

Conformidad. Los ingenieros de software no son los únicos que encaran la complejidad. Los físicos tratan con objetos terriblemente complejos incluso a nivel de partículas "fundamentales". Sin embargo, el trabajo del físico descansa en una sólida fe de que existen principios unificadores a ser descubiertos, ya sea en los quarks o en las teorías del campo unificado. Einstein con frecuencia argumentaba que debían existir explicaciones simplificadas de la naturaleza, porque dios no es caprichoso o arbitrario.

No hay tal fe que consuele al ingeniero de software. Gran parte de la complejidad que debe llegar a gestionar tiene una complejidad arbitraria, forzada sin rima o razón por muchas tradiciones humanas y sistemas a los que sus interfaces deben ajustarse. Estos difieren de una interfaz a otra, y de tanto en tanto, no debido a la necesidad sino solo porque fueron diseñados por diferentes personas y no por dios.

En muchos casos el software debe adecuarse porque es el último en entrar en escena. En otros debe adecuarse porque se percibe como el más adaptable. Pero en todos los casos, gran parte de la complejidad proviene de su adaptación a otras interfaces; esta complejidad no puede simplificarse por ningún rediseño del software únicamente.

Capacidad de cambio. El software está sujeto constantemente a presiones de cambio. Por supuesto, también los edificios, autos y computadoras. Aunque los articulos manufacturados rara vez cambian después de su fabricación; son reemplazados por modelos posteriores, o por la incorporación de cambios importantes en copias posteriores del mismo diseño básico. El retiro del mercado de automóviles es realmente bastante raro; así también son infrecuentes los cambios en computadoras ya vendidas. Ambos son mucho menos frecuentes que las modificaciones del software ya distribuido.

En parte, esto es porque el software en un sistema encarna su función en sí, y la función es la parte que más siente las presiones de cambio. En parte, también es porque el software puede modificarse más fácilmente – es un artefacto intelectual infinitamente maleable. De hecho los edificios cambian, aunque los altos costos del cambio todos lo entienden y sirven para amortiguar los caprichos por cambios.

Todo software exitoso inexorablemente cambia. Aquí suceden dos procesos. Cuando un producto de software resulta útil, las personas lo prueban

en nuevos casos al borde, o más allá, del dominio original. Las presiones para extender su funcionalidad provienen principalmente de los usuarios quienes gustan de la funcionalidad básica e inventan nuevos usos.

En segundo término, el software exitoso también sobrevive más allá de la vida normal de la computadora para la cual fue desarrollada en principio. Si no, llegan nuevas computadoras, o al menos nuevos discos, nuevos desplegadores y nuevas impresoras; y el software debe adaptarse a estos nuevos vehículos de oportunidades.

En resumen, los productos de software forman parte de una matriz cultural de aplicaciones, usuarios, leyes y computadoras. Todos ellos cambian constantemente, y sus cambios obligan inexorablemente al software a cambiar.

Invisibilidad. El software es invisible y no visualizable. Las abstracciones geométricas son herramientas poderosas. El plano de un edificio ayuda tanto al arquitecto como al cliente a evaluar los espacios, el tránsito y las vistas. Las contradicciones llegan a ser obvias y se pueden detectar omisiones. Los dibujos a escala de partes mecánicas y los modelos de palitos de moléculas, aunque son abstracciones, sirven al mismo propósito. Una realidad geométrica es aprehendida a través de una abstracción geométrica.

La realidad del software no forma parte intrínseca del espacio. Por lo tanto no tiene una representación geométrica directa en la forma en que existen mapas de la tierra, en que los chips de silicio tienen diagramas y las computadoras tienen esquemas de conexiones. Tan pronto como intentamos hacer el diagrama de una estructura de software, encontraremos que está compuesto, no de uno, sino de varios grafos dirigidos generales, superpuestos unos sobre otros. Estos grafos pueden representar flujos de control, flujos de datos, patrones de dependencia, secuencias de tiempo o relaciones nombre-espacio. Generalmente no son ni siquiera planos, mucho menos jerárquicos. Por cierto, una de las formas de establecer el control conceptual sobre tal estructura es forzarla a reducir los enlaces hasta que uno o más grafos lleguen a ser jerárquicos.[2]

A pesar del progreso en restringir y simplificar las estructuras del software, aún permanecen básicamente no visualizables, privando así a la mente de una de las herramientas conceptuales más poderosas. Esta carencia no

solo impide el proceso de diseño dentro de una sola mente, sino que dificulta severamente la comunicación entre nuestras mentes.

Los Adelantos del Pasado Resolvían Dificultades Accidentales

Si examinamos los tres pasos que han sido más fructíferos en la tecnología del software en el pasado, descubrimos que cada uno de ellos atacó una dificultad mayor diferente en la construcción del software, aunque estas dificultades fueron accidentales, no esenciales. También podemos observar los límites naturales a la extrapolación de cada uno de tales ataques.

Los lenguajes de alto nivel. Con seguridad el impacto más importante para la productividad, confiabilidad y simplicidad del software ha sido el uso creciente de lenguajes de programación de alto nivel. La mayor parte de los observadores le acreditan a ese desarrollo un aumento de al menos un factor de cinco en la productividad, y con ganancias correspondientes en confiabilidad, simplicidad e inteligibilidad.

¿Qué consigue el lenguaje de alto nivel? Liberar al programa de gran parte de su complejidad accidental. Un programa abstracto consta de construcciones conceptuales: operaciones, tipos de datos, secuencias y comunicación. Un programa de computadora concreto está involucrado con bits, registros, condiciones, bifurcaciones, canales, discos y demás. En la medida en que el lenguaje de alto nivel plasme las construcciones deseadas en el programa abstracto y evite todos los de bajo nivel, elimina un nivel completo de complejidad que jamás fue intrínseco al programa en absoluto.

Lo más que un lenguaje de alto nivel puede hacer es suministrar todas las construcciones que el programador imagina en el programa abstracto. Ciertamente, el nivel de sofisticación de nuestro pensamiento acerca de estructuras de datos, tipos de datos, y operaciones está aumentando constantemente, aunque a un ritmo cada vez menor. Además, el desarrollo de los lenguajes se aproxima cada vez más a la sofisticación de los usuarios.

Además, se llega a un punto en que la elaboración de un lenguaje de alto nivel llega a ser una carga que incrementa, no reduce, la labor intelectual del usuario que rara vez utiliza las construcciones más esotéricas.

Tiempo compartido. La mayoría de los observadores le acreditan al tiempo compartido un gran aumento en la productividad de los programadores y en la calidad de sus productos, aunque no tan grande como la conseguida por los lenguajes de alto nivel.

El tiempo compartido ataca una dificultad claramente diferente. El tiempo compartido preserva la inmediatez, y por lo tanto nos hace capaces de mantener una visión global de la complejidad. La lenta velocidad de respuesta de la programación por lotes significa que inevitablemente olvidamos las minucias, si no el mismo impulso, de lo que estamos pensando cuando dejamos de programar y requerimos compilar y ejecutar. Esta interrupción de la conciencia es costosa en tiempo, porque debemos refrescar nuestra memoria. El efecto más serio puede ser la pérdida de comprensión de todo lo que sucede en un sistema complejo.

La lenta velocidad de respuesta, como las complejidades del lenguaje de máquina, es una dificultad accidental en vez de esencial del proceso de software. Los límites de la contribución del tiempo compartido se derivan directamente. El efecto principal es la reducción del tiempo de respuesta del sistema. En tanto nos aproximamos a cero, en cierto punto se pasará el umbral de perceptibilidad humana, alrededor de 100 milisegundos. Mas allá de este umbral no se esperan más beneficios.

Ambientes integrados de desarrollo. Unix e Interlisp han sido los primeros ambientes de programación integrados utilizados ampliamente, y parecen haber mejorado la productividad por factores enteros. ¿Por qué?

Ambos atacan las dificultades accidentales al usar programas individuales en forma *conjunta*, proporcionan bibliotecas integradas, formatos unificados de archivos, tuberías y filtros. En consecuencia, las estructuras conceptuales que en principio siempre podían invocarse, alimentarse y usarse entre sí, pueden en efecto hacerlo fácilmente en la práctica.

Este avance a su vez ha estimulado el desarrollo de todo un banco de herramientas, puesto que cada nueva herramienta podría ser aplicada a cualquier programa que utilice los formatos estándar.

Debido a estos éxitos, los ambientes son la materia de gran parte de la investigación actual en ingeniería de software. En la siguiente sección veremos sus promesas y limitaciones.

Esperanzas por la Plata

Ahora veremos los desarrollos técnicos que habitualmente se consideran como potenciales balas de plata. ¿Qué problemas abordan – son los problemas de la esencia, o son los residuos de nuestras dificultades accidentales? Ofrecen avances revolucionarios o solo incrementales.

Ada y otros avances en los lenguajes de alto nivel. Uno de los desarrollos recientes más promocionados es el lenguaje de programación Ada, un lenguaje de alto nivel, de propósito general de los 80's. Ada, en efecto, no sólo refleja mejoras evolutivas en los conceptos del lenguaje, sino que plasma rasgos que promueven conceptos de diseño moderno y modularización. Quizá la filosofía de Ada sea más avanzada que el propio lenguaje Ada, debido a su filosofía de modularización, de tipos de datos abstractos y de estructuras jerárquicas. Ada es quizás sobreabundante, el resultado natural del proceso por el cual hubo una axageración de requisitos en su diseño. Eso no es fatal, los problemas de aprendizaje pueden ser resueltos por el subconjunto de vocabularios de trabajo, y el progreso del hardware nos dará MIPS baratos para pagar los costos de compilación. Avanzar en la estructuración de los sistemas de software son en efecto un muy buen uso para el incremento de MIPS que nuestros dólares pagarán. Los sistemas operativos, cuyo desprestigio fue notorio en los 60's por su consumo de memoria y ciclos de reloj, han demostrado ser una forma excelente en los cuales usar algunos de los MIPS y bytes de memoria baratos de la pasada oleada de hardware.

Sin embargo, Ada no demostrará ser la bala de plata que asesine al monstruo de la productividad del software. Después de todo, es sólo otro lenguaje de alto nivel, y la mayor recompensa de tales lenguajes provino de la primera transición, pasando de las complejidades accidentales del lenguage de máquina a declaraciones más abstractas de soluciones graduales. Una vez que se hayan eliminado esos accidentes los restantes serán más pequeños, y los beneficios obtenidos por su remoción seguramente serán menores.

Predigo que dentro de una década, cuando se evalúe la efectividad de Ada, se verá que realizó una diferencia substancial, pero no por algún rasgo particular del lenguaje, ni siquiera debido a la combinación de todos ellos.Tampoco el nuevo ambiente de Ada probará ser la causa de las mejoras. La mayor contribución de Ada será que al haberse cambiado a este lenguaje produjo la ca-

pacitación de los programadores en técnicas modernas de diseño de software.

La programación orientada a objetos. Muchos practicantes del área guardan más esperanzas en la programación orientada a objetos que en cualquier otra de las técnicas de moda.[3] Yo me encuentro entre ellos. Mark Sherman de Darmouth hace notar que debemos tener cuidado al distinguir dos ideas separadas que van bajo ese nombre: tipos abstractos de datos y tipos jerárquicos, también llamados *clases.* El concepto de tipo abstracto de datos es que el tipo de un objeto debe estar definido por un nombre, un conjunto apropiado de valores, y un conjunto apropiado de operaciones, más que por una estructura de almacenamiento, la cual debería estar oculta. Algunos ejemplos son los paquetes de Ada (con tipos privados) o los módulos de Modula.

Los tipos jerárquicos, tales como las clases de Simula-67, permiten la definición de interfaces generales que además pueden ser refinadas por el suministro de tipos subordinados. Los dos conceptos son ortogonales – pueden existir jerarquías sin ocultamiento y ocultamiento sin jerarquías. Ambos conceptos representan avances reales en el arte de la construcción del software.

Cada uno de ellos elimina una dificultad accidental más del proceso, y permite al diseñador expresar la esencia de su diseño sin tener que expresar grandes cantidades de material sintáctico que no añade nuevo contenido de información. Para los tipos tanto abstractos como jerárquicos, la consecuencia es eliminar un tipo de dificultad accidental de orden superior y permitir una expresión de orden superior del diseño.

Sin embargo, tales avances no pueden hacer más que eliminar todas las dificultades accidentales de la expresión del diseño. La complejidad del diseño en sí es esencial; y dichos ataques no producen ningún cambio al respecto. La programación orientada a objetos puede obtener un incremento de un orden de magnitud sólo si la maleza innecesaria de la especificación de tipos que permanece actualmente en nuestros lenguajes de programación sea por sí misma responsable de nueve décimas partes del trabajo involucrado en el diseño de los productos de programación. Yo lo dudo.

Inteligencia artificial. Mucha gente espera que los avances en inteligencia artificial proporcionen el gran paso revolucionario que dará ganancias de órdenes de magnitud a la productividad y a la calidad del software.[4] Yo no.

Para ver por qué, debemos diseccionar qué se entiende por "inteligencia artificial" y entonces veremos cómo se aplica.

Parnas ha aclarado el caos terminológico:

> *Actualmente existen dos definiciones de uso común bastante diferentes de la IA. IA-1: El uso de computadoras para resolver problemas que previamente solo se podían resolver a través de la aplicación de la inteligencia humana. IA-2: El uso de un conjunto específico de técnicas de programación conocidas como heurísticas o programación basada en reglas. En este enfoque se estudia a humanos expertos para determinar qué heurísticas o reglas empíricas usan para resolver problemas. . . . Se diseña el programa para resolver un problema a la manera en que los humanos parecen hacerlo.*
> *La primera definición tiene un significado movedizo. . . . Actualmente, ciertas cosas pueden encajar con la definición de la IA-1 pero, una vez que observamos cómo funciona el programa y entendemos el problema, no lo consideraremos más como IA. . . . Lamentablemente no puedo identificar un cuerpo de tecnología que sea exclusivo de este campo. . . . La mayor parte del trabajo es específico del problema, y se requiere cierta abstracción o creatividad para ver cómo transferirlo.*[5]

Estoy completamente de acuerdo con esta crítica. Las técnicas usadas en el reconocimiento de voz parecen tener poco en común con aquellas utilizadas en el reconocimiento de imágenes, y ambas son diferentes de las utilizadas en los sistemas expertos. He tenido dificultades en ver cómo el reconocimiento de imágenes, por ejemplo, haga un aporte considerable en la práctica de la programación. Y lo mismo con el reconocimiento de voz. La dificultad de construir software es decidir qué se quiere expresar, y no cómo expresarlo. Ningún medio de expresión puede brindar más que incrementos marginales.

La tecnología de los sistemas expertos, IA-2, merece una sección aparte.

Sistemas expertos. La parte más avanzada y más ampliamente aplicada de la inteligencia artificial, es la tecnología para la construcción de sistemas expertos. Muchos científicos de software trabajan arduamente aplicando esta tecnología a ambientes de construcción de software.[5] ¿Cuál es el concepto, y cuáles son sus perspectivas?

Un sistema experto es un programa que tiene un motor de inferencia generalizada y una base de reglas. Está diseñado para tomar datos de entrada y suposiciones, y explora las consecuencias lógicas a través de inferencias derivadas de la base de reglas, obtiene conclusiones y consejos, y ofrece explicar sus resultados mostrando al usuario la trazabilidad de su razonamiento. Los motores de inferencia normalmente pueden tratar con datos difusos o probabilísticos y reglas, además de la lógica puramente determinista.

Tales sistemas ofrecen algunas ventajas claras sobre los algoritmos programados al obtener las mismas soluciones a los mismos problemas:

- La tecnología de los motores de inferencia se desarrolló en forma de una aplicación independiente y solo luego se aplicó a diversos usos. Se puede justificar así una mayor inversión en los motores de inferencia. De hecho, esa tecnología está bastante avanzada.
- Las partes variables de los materiales específicos de la aplicación se codifican en la base de reglas de una forma uniforme, y existen herramientas para desarrollar, cambiar, probar y documentar esta base de reglas. Esto estandariza gran parte de la complejidad de la aplicación misma.

Edward Feigenbaum afirma que el poder de tales sistemas no proviene de los mecanismos de inferencia siempre extravagantes, sino más bien de la base de conocimiento siempre rica que refleja el mundo real de forma más exacta. Creo que el avance más importante que ofrece esta tecnología consiste en la separación de la complejidad de la aplicación del programa en sí.

¿Cómo se puede aplicar esto a la tarea del software? De muchas maneras: sugerir reglas de interfaz, recomendar acerca de estrategias de pruebas, recordar las frecuencias del tipo de errores, ofrecer consejos acerca de la optimización, etc.

Por ejemplo, consideremos un consejero de prueba imaginario. En su forma más rudimentaria, el diagnóstico de los sistemas experto es muy parecido a la lista de control de un piloto, ofrece básicamente sugerencias como las posibles causas del problema. A medida que la base de reglas se desarrolla, las sugerencias se hacen más específicas, toma en cuenta de manera más sofisticada los informes de los síntomas del problema. Uno lo puede ver como un auxiliar de depuración que ofrece sugerencias muy generales al principio, pero a medida que se incorpora cada vez más la estructura del sistema en la base de reglas, llega a ser cada vez más específico en las hipótesis que genera y en las pruebas que recomienda. Dicho sistema experto puede diferir muy radicalmente de los convencionales en que su base de reglas probablemente debería estar modularizada jerárquicamente de la misma forma que los producto de software correspondientes, así que a medidad que el producto se modifica modularmente, la base de reglas de diagnóstico también puede modificarse modularmente.

El trabajo necesario para generar reglas de diagnóstico es un trabajo que deberá hacerse de cualquier manera para generar el conjunto de casos de prueba para los módulos y el sistema. Si se hizo de una forma adecuada y general, con una estructura uniforme para las reglas y el acceso a un buen motor de inferencia, podría realmente reducirse el trabajo entero de generar casos de prueba ilustrativos, además como ayuda al mantenimiento de por vida y en las pruebas de modificaciones. De la misma forma, podemos proponer otros consejeros – probablemente muchos y quizás simples – aplicables a otras partes de la tarea de construcción del software.

Muchas dificultades se interponen en el camino del programador en la ralización temprana de consejeros expertos útiles. Una parte crucial de nuestro escenario imaginario es el desarrollo de formas fáciles de obtener a partir de la especificación de la estructura del programa la generación automática o semiautomática de reglas de diagnóstico. Aún más difícil e importante es la doble tarea de la adquisición de conocimiento: encontrar expertos elocuentes, autoanalíticos que sepan *por qué* hacen las cosas; y desarrollar técnicas eficientes para la extracción de lo que ellos conocen y su destilación en forma de bases de reglas. El prerequisito esencial en la construcción de un sistema experto es tener un experto.

La mayor contribución de los sistemas expertos seguramente será la de

poner al servicio del programador inexperto la experiencia y la sabiduría acumulada de los mejores programadores. Este aporte no es menor. La brecha entre la mejor práctica del ingeniero de software y la práctica promedio es muy amplia – quizás más amplia que en otras ramas de la ingeniería. Una herramienta que difunda una buena práctica sería algo importante.

Programación "automática". Por casi 40 años, la gente ha estado anticipando y escribiendo acerca de la "programación automática,", o la generación de un programa para resolver un problema a partir de la exposición de las especificaciones del problema. Hoy, algunas personas escriben como si esta tecnología fuese a proporcionar el siguiente gran salto.[7]

Parnas da a entender que el término se ha utilizado por el glamour y no por el contenido semántico, afirma,

> *En resumen, la programación automática siempre ha sido un eufemismo para la programación con un lenguaje de más alto nivel del que disponía el programador en ese momento.*[3]

En esencia, argumenta que en la mayoría de los casos es el método de solución, no el problema, lo que requiere ser especificado.

Se pueden encontrar excepciones. La técnica para construir generadores es muy poderosa, y se usa rutinariamente con buenos resultados en programas de clasificación. Algunos sistemas para la integración de ecuaciones diferenciales también han permitido la especificación directa del problema. El sistema evaluaba los parámetros, seleccionados de una biblioteca de métodos de solución y generaba los programas.

Estas aplicaciones tienen propiedades muy convenientes:

- Los problemas se caracterizan rápidamente mediante unos pocos parámetros.
- Existen muchos métodos conocidos de solución que proporcionan una biblioteca de opciones.
- Un análisis extensivo ha conducido a reglas explícitas para seleccionar las técnicas de solución, dados los parámetros del problema.

Es difícil ver cómo generalizar dichas técnicas a un universo más amplio de los sistemas de software comunes, donde los casos con propiedades tan claras son la excepción. Es difícil incluso imaginar cómo podría probablemente suceder este gran avance en la generalización.

Programación gráfica. Un tema favorito para las disertaciones de doctorado en ingeniería de software es la programación gráfica, o visual, esto es, la aplicación de gráficas de computadora al diseño de software.[9] Algunas veces la promesa de tal enfoque se postula por la analogía con el diseño de chips VLSI, donde las gráficas de computadora juegan un papel fundamental. A veces este enfoque está justificado al considerar los diagramas de flujo como el medio ideal de diseño de programas, y proporcionan medios poderosos para su construcción.

Nada convincente, ni mucho menos emocionante, aún ha surgido de dichos esfuerzos. Estoy convencido de que nada surgirá.

En primer lugar, como he argumentado en otro lado, el diagrama de flujo es una abstracción muy pobre de la estructura del software.[10] Y en efecto, es mejor verlo como un intento desesperado de dotar de un lenguaje de control de alto nivel a la computadora propuesta por Burks, von Neumann y Goldstine. La lastimosa forma de elaborar diagramas de flujo hoy en día, a través de múltiples páginas y de cajas de conexiones, ha demostrado ser esencialmente inútil como herramienta de diseño – los programadores dibujan diagramas de flujo después de, no antes de, escribir los programas que los describen.

En segundo lugar, las pantallas de hoy son muy pequeñas, en píxeles, para mostrar tanto el alcance como la resolución de cualquier detalle importante del diagrama de software. La así llamada "metáfora de escritorio" de las estaciones de trabajo actuales es en su lugar una metáfora de un "asiento de avión." Cualquiera que haya revuelto un montón de papeles mientras está sentado entre dos pasajeros corpulentos reconocerá la diferencia – solo se pueden ver unas cuantas cosas a la vez. El escritorio real proporciona una visión global y accesos aleatorios a una veintena de páginas. Mas aún, cuando se produce un fuerte ataque de creatividad, más de un programador o escritor se ha sabido que abandona su escritorio por un espacio más amplio. La tecnología del hardware tendrá que avanzar bastante antes de que el ámbito global de nuestros ámbitos particulares sea suficiente para la tarea del

diseño de software.

Tal como he argumentado anteriormente, el software en esencia es muy difícil de visualizar. Ya sea que dibujemos flujos de control, anidamientos de alcance variable, referencias cruzadas variables, flujos de datos, estructuras de datos jerárquicas, o lo que sea, sentimos solo una dimensión del intrincado entrelazado del elefante del software. Si superponemos todos los diagramas generados por las muchas vistas relevantes, es difícil extraer alguna visión global. La analogía con el VLSI es esencialmente engañosa – el diseño de un chip es un objeto tendido en dos dimensiones cuya geometría refleja su esencia. Un sistema de software no lo es.

Verificación de programas La mayor parte del trabajo en la programación moderna se dedica a probar y a reparar errores. ¿Hallaremos quizá una bala de plata cuando eliminemos los errores en el código fuente en la fase de diseño del sistema? ¿Podremos mejorar radicalmente tanto la productividad como la confiabilidad del producto siguiendo la estrategia totalmente distinta de proporcionar diseños correctos antes de volcar un inmenso esfuerzo en la implementación y la prueba de los mismos?

Dudo que aquí encontremos la magia. La verificación de programas es un concepto bastante poderoso, y será muy importante para cosas tales como la seguridad en los núcleos de los sistema operativos. Sin embargo, la tecnología no promete ahorrarnos trabajo. Las verificaciones representan tanto trabajo que solo un puñado de programas substanciales son verificados.

La verificación de programas no significa programas a prueba de errores. Aquí tampoco hay magia. Las demostraciones matemáticas también están sujetas a errores. Así que mientras las verificaciones podrían reducir la carga de prueba del programa, no podrán eliminarlas.

Más seriamente, incluso la verificación perfecta del programa puede sólo establecer que un programa cumple con sus especificaciones. La parte más difícil de la labor de software es llegar a una especificación completa y coherente, y gran parte de la esencia de construir un programa es de hecho la depuración de las especificaciones.

Ambientes y herramientas. ¿Cuánto más podemos esperar de la explosión en las investigaciones acerca de mejores ambientes de programación? La

reacción instintiva es que los problemas con grandes recompensas fueron los primeros en ser atacados, y han sido resueltos: los sistemas de archivos jerárquicos, la uniformidad en los formatos de archivo como en las interfaces entre programas y herramientas generalizadas. Los editores inteligentes de lenguajes específicos son desarrollos que todavía no son utilizados ampliamente en la práctica, aunque la mayoría de ellos promete liberarnos de errores sintácticos y errores semánticos simples.

Quizá el principal beneficio que aún queda pendiente en los ambientes de programación es el uso de sistemas de bases de datos integrados que lleve un seguimiento de la multitud de detalles que cada programador debe recordar exactamente y que mantenga su actualización dentro de un grupo de colaboradores en un mismo sistema.

Sin duda este trabajo vale la pena y dará frutos tanto en la productividad como en la confiabilidad. Aunque por su propia naturaleza, la ganancia de aquí en adelante debe ser marginal.

Estaciones de trabajo. ¿Qué beneficios podemos esperar para el software del inevitable y rápido incremento en la potencia y la capacidad de memoria de las estaciones de trabajo personales? Bueno, ¿Cuántos MIPS se pueden usar de forma fructífera? La redacción y edición de programas y documentos están completamente respaldadas por las velocidades actuales. A la compilación le vendría bien un impulso, aunque un factor de 10 en la velocidad de la máquina seguramente dejaría tiempo para pensar, la principal actividad diaria del programador. De hecho, así parece ser actualmente.

Estaciones de trabajo más poderosas sin duda son bienvenidas. No podemos esperar mejoras mágicas de ellas.

Promesa de Ataque a la Esencia Conceptual

A pesar de que ningún gran avance tecnológico promete dar el tipo de resultados mágicos con los cuales estamos tan acostumbrados en el área del hardware, existe ahora tanto una abundancia de buen trabajo en curso, como la promesa de un progreso sostenido, si bien poco espectacular.

Todos los ataques tecnológicos sobre los accidentes del proceso de soft-

ware están esencialmente limitados por la ecuación de la productividad:

$$Tiempo\ de\ la\ tarea = \sum_i (Frecuencia)_i \times (Tiempo)_i$$

Si, como creo, los componentes conceptuales de la tarea están ahora ocupando la mayor parte del tiempo, entonces ninguna cantidad de acciones sobre los componentes de la tarea que sean únicamente la expresión de los conceptos podrá generar grandes aumentos en la productividad.

Por lo tanto, debemos considerar aquellos ataques que abordan la esencia del problema del software, la formulación de estas estructuras conceptuales complejas. Afortunadamente, algunos de ellos son muy prometedores.

Comprar en contra de construir. La solución más radical posible para construir software es definitivamente no hacerlo.

Cada día esto se vuelve más fácil, en tanto más y más vendedores ofrecen más y mejores productos de software para una impresionante variedad de aplicaciones. Mientras que nosotros, los ingenieros de software, hemos trabajado en la metodología de la producción, la revolución de las computadoras personales ha creado no uno, sino muchos mercados masivos para el software. En cada puesto de periódicos vemos revistas mensuales que anuncian y revisan docenas de productos, clasificadas por tipo de máquina, a precios que van desde unos cuantos dólares a unos cientos. Fuentes más especializadas ofrecen productos muy poderosos para estaciones de trabajo y otros mercados de Unix. Incluso ya podemos adquirir herramientas y ambientes de desarrollo de software. También he propuesto en otros foros un mercado para módulos individuales.

Es más barato comprar cualquiera de esos productos que construirlo de nuevo. Incluso a un precio de $100,000, una pieza de software comprada cuesta aproximadamente lo mismo que el salario anual de un programador. ¡Y la entrega es inmediata! Inmediata al menos para productos que realmente existen y de los cuales el desarrollador puede dar referencias de clientes satisfechos. Mas aún, tales productos tienden a estar mucho mejor documentados y algo mejor mantenidos que el software desarrollado en casa.

Creo que el desarrollo de un mercado masivo es la tendencia más importante a largo plazo en la ingeniería de software. El costo del software ha sido

siempre el costo de su desarrollo, no el costo de su replicación. Compartir estos costos incluso entre unos cuantos usuarios reduce radicalmente el costo por usuario. Otra forma de verlo es que el uso de *n* copias de un sistema de software en efecto multiplica la productividad de sus desarrolladores por *n*. Esto mejora la productividad de la disciplina y de la nación.

La cuestión clave, obviamente, es la aplicabilidad. ¿Puedo usar un paquete disponible ya desarrollado para hacer mi tarea? Aquí ha sucedido una cosa sorprendente. Durante los 50's y 60's, varios estudios mostraron que los usuarios no usarían paquetes ya desarrollado para nóminas, control de inventario, cuentas pendientes, etc. Los requisitos eran tan especializados, que la variación de caso a caso era muy alta. Durante los 80's, encontramos paquetes con una alta demanda y uso generalizado. ¿Qué ha cambiado?

En realidad no fueron los paquetes. Puede ser que sean más generalizados y un tanto más personalizables que los anteriores, pero no mucho. En realidad tampoco las aplicaciones. En todo caso, las necesidades científicas y de negocios actuales son más diversas, y más complicadas que las de hace 20 años.

El gran cambio ha estado en la relación de costo hardware/software. El comprador de una máquina de $2 millones en 1960 sentía que podía pagar $250,000 más por un programa de nómina personalizado, uno que se deslice suavemente y sin perjuicios dentro de un ambiente social hostil hacia las computadoras. Los compradores de máquinas de oficina de $50,000 hoy no pueden posiblemente pagar por programas de nómina personalizados, así que adaptan sus procedimientos de nómina a los paquetes disponibles. Las computadoras ahora son lugares comunes, si no es que muy amadas, tal que las adaptaciones son aceptadas como una cosa natural.

Existen excepciones dramáticas a mis argumentos de que la generalización de los paquetes de software ha cambiado poco a través de los años; las hojas de cálculo y los sistemas de bases de datos sencillos. Estas herramientas poderosas, en retrospectiva tan obvias y con todo de aparición tan tardía, se prestan a múltiples usos, algunos muy poco ortodoxos. Actualmente abundan artículos e incluso libros que muestran cómo abordar tareas inesperadas con las hojas de cálculo. Grandes cantidades de aplicaciones que anteriormente hubieran sido escritas como programas personalizados en Cobol o por un Programa Generador de Informes hoy son realizadas rutinariamente con

estas herramientas.

Actualmente muchos usuarios manejan día tras día sus propias computadoras en una variedad de aplicaciones sin jamás haber escrito un programa. De hecho, muchos de ellos no saben escribir nuevos programas para sus computadoras, pero sin embargo son expertos en resolver nuevos problemas con ellas.

Creo que la estrategia de productividad del software más poderosa para muchas organizaciones hoy en día es equipar a trabajadores intelectuales inexpertos en computadoras que laboran en la línea de fuego con computadoras personales y buenos pogramas generalizados de edición de texto, dibujo, manejo de archivos, y hojas de cálculo, y dejarlos libres. La misma estrategia funcionaría si se dotaran de paquetes generalizados de matemáticas y estadística y algunas simples capacidades de programación a cientos de científicos de laboratotorio.

Refinamiento de requisitos y prototipos rápidos. La parte más difícil de la construcción de los sistemas de software es precisamente decidir qué construir. No hay otra parte del trabajo conceptual que sea tan dificil como el establecimiento de requisitos técnicos detallados, incluyendo todas las interfaces tanto a los usuarios, como a las computadoras y a otros sistemas de software. O que perjudique tanto el sistema resultante si se hizo mal. Ni que sea más difícil de rectificar posteriormente.

Por lo tanto, la tarea más importante que los constructores de software hacen por sus clientes es la extracción y el refinamiento iterativo de los requisitos del producto. La verdad es que los clientes no saben lo que quieren. Generalmente no saben qué preguntas se deben responder, y casi nunca han pensado en el problema en esos detalles que deben especificarse. Incluso la respuesta sencilla – "Hacer que el nuevo sistema de software trabaje como nuestro viejo sistema manual de procesamiento de información" – es de hecho bastante simple. Los clientes jamás quieren exactamente eso. Más aún, los complejos sistemas de software son cosas que actúan, que mueven cosas y que funcionan. Las dinámicas de esa acción son difíciles de imaginar. Así que al planificar cualquier tarea de software, es necesario contar con una amplia iteración entre el cliente y el diseñador como parte de la definición del sistema.

Yo iría un paso más lejos y afirmaría que es realmente imposible para los

clientes, incluso para aquellos que trabajan con los ingenieros de software, especificar completa y correctamente los requisitos exactos de un producto de software antes de construir y probar algunas versiones previas del producto que están especificando.

Por lo tanto, uno de los esfuerzos más prometedores de la tecnología actual, y que ataca la esencia, no los accidentes, del problema del software, es el desarrollo de estrategias y herramientas para la obtención rápida de prototipos de sistemas como parte de la especificación iterativa de requisitos.

Un prototipo de un sistema de software es aquel que simula las interfaces importantes y realiza las funciones principales del sistema previsto, sin estar necesariamente sujeto a las mismas restricciones de velocidad del hardware, el tamaño o el costo. Los prototipos generalmente realizan las tareas principales de la aplicación, aunque no intentan manejar excepciones, responder correctamente a entradas inválidas, abortar limpiamente, etc. El propósito del prototipo es hacer realidad la estructura conceptual especificada, de tal manera que el cliente pueda probar su coherencia y su facilidad de uso.

Actualmente gran parte de los procedimientos de adquisición de requisitos de software yacen en la suposición de que se puede especificar un sistema satisfactoriamente por adelantado, obtener ofertas para su construcción, construirlo e instalarlo. Creo que esta suposición está esencialmente errada, y que muchos de los problemas de adquisición de requisitos de software surgen de esa falacia. Por lo tanto, no pueden ser reparados sin una revisión fundamental, una que mantenga el desarrollo y la especificación iterativa de prototipos y productos.

Desarrollo incremental – desarrolle software, no lo construya. Todavía recuerdo la impresión que sentí en 1958 cuando escuché por primera vez a un amigo hablar acerca de *construir* un programa, en oposición a *escribirlo*. En un destello ensanchó toda mi visión del proceso del software. El cambio de metáfora fue poderoso, y exacto. Hoy entendemos en qué se parece la construcción del software a otros procesos de construcción, y abiertamente usamos otros elementos de la metáfora, tales como *las especificaciones, el ensamblado de componentes y el andamiaje.*

La metáfora de la construcción ha sobrevivido a su utilidad. Es tiempo de cambiar otra vez. Si, como creo, las estructuras conceptuales que construi-

mos hoy en día son bastante complicadas para ser especificadas exactamente por adelantado, y muy complejas para ser construidas sin fallas, entonces debemos tomar un enfoque radicalmente diferente.

Volvamos nuestra mirada a la naturaleza y estudiemos la complejidad de las cosas vivas en lugar de sólo la obra inerte del hombre. Aquí encontramos construcciones cuyas complejidades nos estremecen de asombro. Solo el cerebro es intrincado por encima del mapeo, poderoso más allá de la imitación, rico y diverso, autoprotegido y autorenovado. El secreto es que se desarrolla, no se construye.

Así debe ser con nuestros sistemas de software. Hace algunos años Harlan Mills propuso que cualquier sistema de software debe desarrollarse a través de un desarrollo incremental.[11] Es decir, primero debemos construir un sistema para ser ejecutado, aún cuando no haga nada útil excepto llamar al conjunto apropiado de subprogramas ficticios. Luego, se expande poco a poco con subprogramas que están siendo, a su vez, desarrollados dentro de acciones o llamadas a programas aún incompletos y vacíos de bajo nivel.

He visto el resultado más dramático desde que empecé a insistir en esta técnica de construcción de proyectos en mi clase de laboratorio de ingeniería de software. Nada en la década pasada ha cambiado tan radicalmente mi propia práctica o su eficacia. El enfoque necesita un diseño descendente, porque es un desarrollo descendente del software. Esto permite retroceder fácilmente. Se presta para prototipos tempranos. Cada función que se añade y nuevo suministro de datos o circunstancias más complejas nace orgánicamente a partir de lo que allí ya existe.

Los efectos morales son asombrosos. El entusiasmo irrumpe cuando hay un sistema funcionando, aunque sea uno simple. Los esfuerzos se redoblan cuando la primera imagen de un nuevo sistema de software gráfico aparece en la pantalla, aunque sea solo un rectángulo. En cada etapa del proceso, siempre se tiene un sistema funcionando. He hallado que los equipos pueden *desarrollar* entidades mucho más complejas en cuatro meses de las que pueden *construir*.

Se pueden obtener los mismos beneficios en proyectos grandes como en pequeños.[12]

Grandes diseñadores. La principal cuestión de cómo mejorar los centros de desarrollo de software, ha sido como siempre, el personal.

Podemos obtener buenos diseños siguiendo buenas prácticas en lugar de las malas. Las buenas prácticas de diseño se pueden enseñar. Los programadores están entre la porción más inteligente de la población, así que pueden aprender las buenas prácticas. Es por eso que un impulso importante en los Estados Unidos ha sido promulgar buenas prácticas modernas. Nuevos planes de estudio, nueva literatura, nuevas organizaciones como el Instituto de Ingeniería de Software, todos ellos se han creado para mejorar el nivel de nuestra práctica, del nivel malo al bueno. Esto es totalmente apropiado.

Sin embargo, no creo que podamos dar el gran salto adelante de la misma forma. En tanto la diferencia entre los malos diseños conceptuales y buenos dependa de la solidez de los métodos de diseño, la diferencia entre los buenos diseños y los grandes seguramente no. Los grandes diseños provienen de grandes diseñadores. La construcción del software es un proceso *creativo*. Una sólida metodología puede facultar y liberar la mente creativa; no puede avivar o inspirar el trabajo rutinario.

Las diferencias no son menores – se parece mucho a Salieri y Mozart. Después de muchos estudios se ha demostrado que los mejores diseñadores producen estructuras que son más rápidas, más pequeñas, más simples, más limpias y se llevan a cabo con menor esfuerzo. Las diferencias entre el gran enfoque y el promedio son de un orden de magnitud.

Haciendo una pequeña retrospectiva vemos que aunque muchos sistemas de software finos y útiles han sido diseñados por comités y construidos por proyectos de diversas partes, los sistemas de software que han emocionado a apasionados admiradores son aquellos que han sido el producto de una o unas cuantas y grandes mentes diseñadoras. Considere Unix, APL, Pascal, Modula, la interfaz de Smalltalk, incluso Fortran; y lo contrastamos con Cobol, PL/I, Algol, MVS/370, y MS-DOS (Fig. 16.1).

Por lo tanto, aunque apoyo firmemente la transferencia de tecnología y los esfuerzos de desarrollo curricular actualmente en curso, creo que el esfuerzo individual más importante que podemos organizar es el desarrollo de formas de crecimiento de grandes diseñadores.

Ninguna organización de software puede soslayar este reto. Los buenos gestores, por escasos que sean, no son más escasos que los buenos diseñadores.

Sí	**No**
Unix	Cobol
APL	PL/I
Pascal	Algol
Modula	MVS/370
Smalltalk	MS-DOS
Fortran	

Fig. 16.1 Productos emocionantes

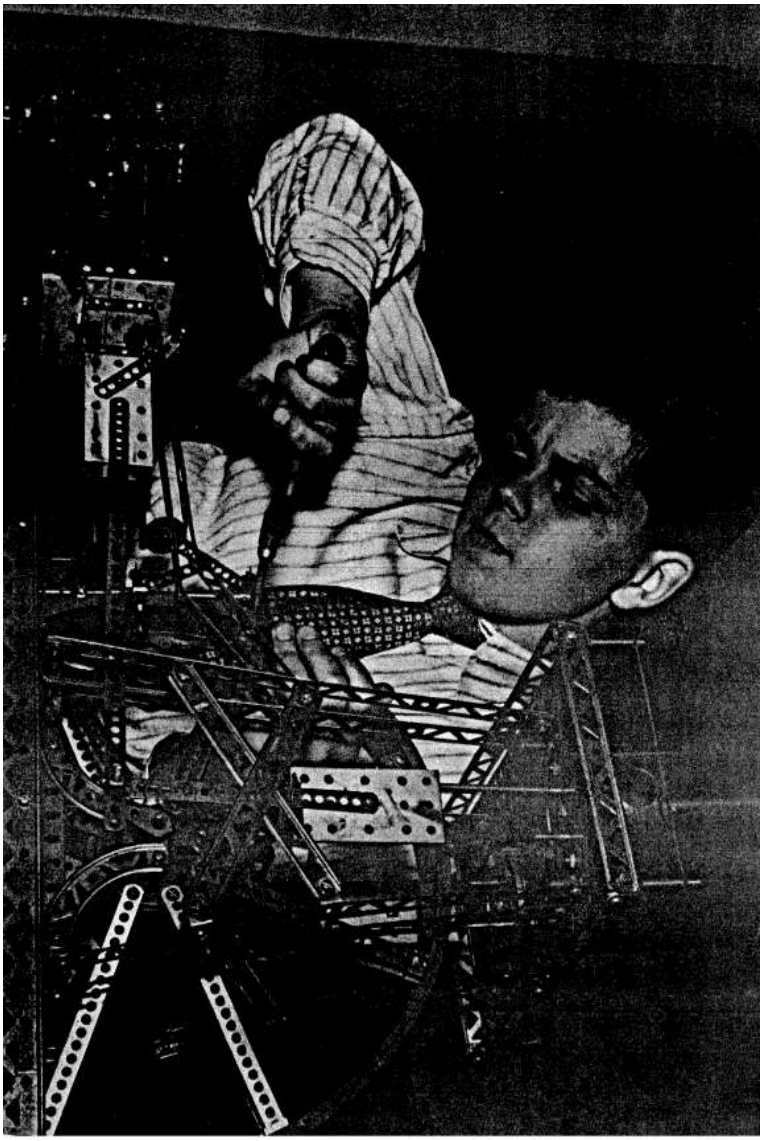
Los grandes diseñadores como los grande gestores son ambos muy raros. La mayoría de las organizaciones invierten un esfuerzo considerable en buscar y desarrollar las perspectivas de gestión; No sé de ninguna que invierta el mismo esfuerzo en buscar y desarrollar a esos grandes diseñadores de los que finalmente dependerá la excelencia técnica de los productos.

Mi primera propuesta es que cada organización de software debe determinar y proclamar que los grandes diseñadores son tan importantes para su éxito como lo son los gestores, y que pueden esperar ser igualmente promovidos y recompensados. No solo el salario, sino los beneficios del reconocimiento – tamaño de la oficina, mobiliario, equipamiento técnico personal, fondos de viaje y personal de apoyo – deben ser totalmente equivalentes.

¿Cómo se desarrolla un gran diseñador? El espacio no nos permite una discusión amplia, aunque algunos pasos son obvios:

- Identificar sistemáticamente a los mejores diseñadores tan pronto como sea posible. Los mejores generalmente no son los más experimentados.
- Asignar un tutor profesional como responsable del desarrollo del candidato, y mantener un meticuloso archivo profesional.
- Idear y mantener un plan de desarrollo profesional de cada candidato, incluyendo la formación cuidadosamente seleccionada con los mejores diseñadores, capítulos de educación formal avanzada y cursos cortos, todo entremezclado con un diseño individual y deberes de liderazgo técnico.
- Proporcionar oportunidades para que los diseñadores en desarrollo interactúen y se estimulen entre ellos.

17
“No Existen Balas de Plata” Recocido



17
"No Existen Balas de Plata"
Recocido

Cada bala tiene su lugar.

GUILLERMO III DE INGLATERRA, PRÍNCIPE DE ORANGE

Quien piense que ve una pieza impecable
Piense que nunca fue, no es, ni será jamás.

ALEXANDER POPE, UN ENSAYO SOBRE EL CRITICISMO

Ensamblaje de una estructura a partir de partes hechas, 1945
The Bettman Archive

Acerca de Hombres Lobo y Otras Historias de Terror

"No Existen Balas de Plata – Lo Esencial y lo Accidental en la Ingeniería de Software" (ahora capítulo 16) fue originalmente un artículo invitado para el congreso del IFIP '86 en Dublín, y fue publicado en esas memorias.[1] La revista *Computer* lo reimprimió, detrás de una cubierta gótica, ilustrado con fotogramas de películas como *El Hombre Lobo de Londres.*[2] También incluyeron una franja lateral explicativa "Para asesinar al hombre Lobo," exponiendo la leyenda (moderna) de que sólo una bala de plata lo haría. No estuve enterado de la franja lateral ni de las ilustraciones antes de su publicación, y jamás esperé que un artículo técnico serio fuera tan adornado.

Sin embargo, los editores de *Computer* eran expertos en lograr el efecto deseado y mucha gente parece haber leído el artículo. Por lo tanto he tenido que escoger aún otra imagen del hombre lobo para ese capítulo, una representación antigua de una criatura casi cómica. Espero que esta imagen menos llamativa tenga el mismo efecto beneficioso.

Existe También una Bala de Plata – ¡Y AQUÍ ESTÁ!

"No Existen Balas de Plata" afirma y argumenta que ningún desarrollo de la ingeniería de software producirá un incremento de un orden de magnitud en la productividad del software en los próximos 10 años (a partir de la publicación del artículo en 1986). Ya han pasado nueve años de la década, así que es oportuno ver cómo se sostiene esta predicción.

Mientras que *El Mítico Hombre-Mes* generó muchas menciones pero pocos debates, "No Existen Balas de Plata" ha generado artículos de refutación, cartas a editores de revistas, y cartas y ensayos que continúan hasta hoy.[3] La mayoría de ellos atacan el argumento principal de que no existe una solución mágica, y mi opinión inequívoca de que no puede haber una. La mayoría está de acuerdo con la mayoría de los argumentos de "NEBP," pero luego continúan afirmando que existe asimismo una bala de plata para el monstruo del software y que el autor lo ha inventado. Hoy, mientras leo de nuevo las primeras respuestas, no puedo ayudar a aclarar que las panaceas promovidas con tanto vigor en 1986 y 1987 no han tenido los efectos espectaculares alegados.

Compro hardware y software principalmente mediante la prueba del "u-

suario satisfecho"– pláticas con clientes *bona fide* que pagan al contado y que usan el producto y están satisfechos. Del mismo modo, voy a creer sin mayor reparo que una bala de plata se ha materializado cuando un usuario independiente *bona fide* dé un paso adelante y diga, "he usado esta metodología, herramienta o producto, y obtuve un incremento de diez veces en la productividad del software."

Muchos corresponsales han hecho enmendaciones válidas o aclaraciones. Algunos han emprendido un análisis punto por punto y refutado, por lo cual estoy muy agradecido. En este capítulo, compartiré las mejoras y abordaré las refutaciones.

La Escritura Poco Clara Será Mal Interpretada

Algunos escritores muestran que fracasé en aclarar algunos argumentos.

Accidente. El argumento central de "NEBP" está tan claramente mencionado en el Resumen del Capítulo 16 como sé hacerlo. Sin embargo, algunos se han confundido por los términos *accidente* y *accidental,* los cuales siguen una antigua costumbre que se remonta a Aristóteles.[4] Por *accidental,* no quiero decir *que suceda al azar,* ni *por la mala suerte,* sino más cercano a lo *incidental,* o *al accesorio.*

Yo no denigraría la partes accidentales de la construcción del software; en su lugar sigo al dramaturgo Inglés, al escritor de novelas policiacas y a la teóloga Dorothy Sayers al observar que toda actividad creativa consta de (1) la formulación de las construcciones conceptuales, (2) la implementación en un medio real, y (3) la interactividad con los usuarios en aplicaciones reales.[5] La parte de la construcción del software que llamo *esencia* es la elaboración mental de la construcción conceptual; la parte que yo llamo *accidente* es su proceso de implementación.

Una cuestión de hecho. Me parece (aunque no a todos) que la veracidad del argumento central se reduce a una cuestión de hecho: ¿Qué fracción del esfuerzo total del software está asociado hoy en día con la representación exacta y ordenada de la construcción conceptual, y qué fracción con el esfuerzo de la elaboración mental de las construcciones? La búsqueda y reparación de defectos cae parcialmente en cada fracción, de acuerdo a si los defectos son

conceptuales, tales como el fracaso para reconocer una excepción, o de representación, tales como un error en el apuntador o en la asignación de memoria.

Es mi opinión, y nada más, que la parte accidental o de representación del trabajo está ahora debajo de la mitad o menos del total. Puesto que esta fracción es una cuestión de hecho, su valor podría en principio ser establecida a través de mediciones.[6] En su defecto, mi estimación puede ser corregida por estimaciones mejor informadas y más actuales. Es significativo de que nadie haya afirmado por escrito de forma pública o privada que la parte accidental sea tan grande como 9/10.

"NEBP" argumenta, sin discusión, que si la parte accidental del trabajo es inferior a 9/10 del total, reduciéndolo a cero (lo cual sería un milagro) no producirá un incremento en la productividad de un orden de magnitud. Por lo tanto, se *debe* atacar la esencia.

A partir de "NEPB", Bruce Blum ha llamado mi atención al trabajo de Herzberg, Mausner y Sayderman[7] de 1959. Hallaron que los factores motivacionales *pueden* incrementar la productividad. Por otro lado, los factores ambientales y accidentales, no importando cuán positivos sean, no pueden hacerlo; aunque estos factores pueden decrementar la productividad cuando son negativos. "NEPB" argumenta que gran parte del progreso del software ha estado en la remoción de tales factores negativos: lenguajes de máquina sorprendentemente incómodos, procesamiento por lotes con largos tiempos de respuesta, herramientas deficientes y severas restricciones de memoria.

Son, por lo tanto, las dificultades *esenciales irremediables*? "Existe una Bala de Plata,", excelente artículo de 1990 de Brad Cox, argumenta con elocuencia por el enfoque del componente reusable e intercambiable como un ataque a la esencia conceptual del problema.[8] Estoy muy de acuerdo.

Sin embargo, Cox malinterpreta dos punto de "NEPB". Primero, lo entiende como la afirmación de que las dificultades del software surgen "de una deficiencia en cómo los programadores construyen el software hoy en día." Mi argumento fue que las dificultades esenciales son inherentes a la complejidad conceptual de la funcionalidad del software, y no depende de cuándo y qué método se use para construirlo. Segundo, él (y otros) entienden "NEBP" como una afirmación de que no hay esperanza en atacar las dificultades esenciales de la construcción del software. Esa no fue mi intención. La elaboración

de la construcción conceptual tiene de hecho como dificultades intrínsecas la complejidad, conformidad, capacidad de cambio e invisibilidad. Sin embargo, los problemas causados por cada una de estas dificultades pueden mejorarse.

La complejidad se da por niveles. Por ejemplo, la complejidad es la más seria dificultad intrínseca, aunque no toda la complejidad es inevitable. Gran parte, aunque no toda, de la complejidad conceptual de nuestras construcciones de software viene de la complejidad arbitraria de las aplicaciones mismas. En efecto, Lars Sødhal de MYSIGMA Sødhal and Partners, una firma multinacional de gestión, escribe:

> *En mi experiencia la mayor parte de las complejidades que encontramos en el trabajo de sistemas son síntomas de mal funcionamiento organizacional. Tratar de modelar esta realidad con programas igualmente complejos es realmente conservar el desorden en lugar de resolver los problemas.*

Steve Lukasik de Northrop sostiene que incluso la complejidad organizacional tal vez no sea arbitraria, sino que puede ser susceptible a los principios de organización:

> *Me formé como físico y por tanto veo las cosas "complejas" como susceptibles a la descripción en términos de conceptos más simples. Ahora usted puede estar en lo cierto; No afirmaré que todas las cosas complejas son susceptibles a los principios de organización. . . . por las mismas reglas del argumento usted no puede afirmar que ellas puedan* no *serlo.*
>
> *. . . La complejidad de ayer es el orden de mañana. La complejidad del desorden molecular brindó un camino a la teoría cinética de los gases y a las tres leyes de la termodinámica. Ahora, el software puede que ni siquiera revele esos tipos de principios de organización, aunque la carga recae en usted para explicar por qué no. No estoy siendo obtuso o polémico. Creo que algún día la "complejidad" del software será entendida en términos de nociones de mayor orden (*invariantes *para el físico).*

No he emprendido un análisis más profundo del que Lukasik solicita correctamente. Como disciplina, necesitamos una teoría de la información am-

pliada que cuantifique el contenido de información de las estructuras estáticas, así como hace la teoría de Shannon para las comunicaciones. Eso está absolutamente más allá de mis posibilidades. A Lukasik simplemente le respondería que la complejidad del sistema es una función de un sinnúmero de detalles que deben ser especificados con exactitud, ya sea a través de una regla general o detalle a detalle, y no sólo estadísticamente. Parece muy improbable que el trabajo no coordinado de mucha mentes tendría suficiente coherencia para ser descrito exactamente a través de reglas generales.

Sin embargo, gran parte de la complejidad en una construcción de software no se debe a la conformidad con el mundo externo sino más bien a la implementación misma – sus estructuras de datos, sus algoritmos, su conectividad. Desarrollar software en fragmentos de mayor nivel, construido por alguien más o reusado de nuestro propio pasado, evita encarar capas completas de complejidad. "NEBP" aboga por un ataque incondicional a los problemas de la complejidad, pero que se puedan realizar progresos es bastante optimista. Aboga por agregar la complejidad necesaria a un sistema de software:

- Jerárquicamente, a través de la estratificación de módulos u objetos.
- Incrementalmente, de tal manera que el sistema siempre esté funcionando.

Análisis de Harel

David Harel, en un artículo de 1992, "Biting the Silver Bullet," emprende el análisis más cuidadoso de "NEBP" que ha sido publicado.[9]

Pesimismo vs. optimismo vs. realismo. Harel observa tanto "NEPB" como el artículo de Parnas de 1984 "Software Aspects of Strategic Defense Systems"[10] como "demasiado desalentadores." Así que apunta a iluminar el lado más claro de la moneda, subtitulando su artículo "Toward a Brighter Future for System Development." Cox, como también Harel, percibe "NEPB" como pesimista, y dice, "Aunque si usted observa estos mismos hechos desde una nueva perspectiva, emerge una conclusión más optimista." Ambos malinterpretan el sentido.

Primero, mi esposa, mis colegas y mis editores me encuentran pecando con mayor frecuencia de optimismo que de pesimismo. Después de todo, soy un programador de origen, y el optimismo es una enfermedad profesional de nuestro oficio.

"NEPB" explícitamente dice "Como se ve el horizonte de aquí a una década, no vemos ninguna bala de plata . . . Sin embargo, ser escéptico no implica ser pesimista . . . No hay un atajo, pero hay un camino." Predice que las innovaciones en curso en 1986, en caso de ser desarrolladas y explotadas, *juntas* conseguirían en efecto un incremento de un orden de magnitud en la productividad. A medida que avanza la década de 1986 a 1996, esta predicción parece, si cabe, muy optimista en lugar de muy pesimista. Incluso si "NEBP" fuera universalmente visto como pesimista, ¿Qué hay de malo en ello? Es la afirmación de Einstein de que nada puede viajar mas rápido que la velocidad de la luz "desolador" o "pesimista?" ¿Qué decir del resultado de Gödel de que algunas cosas no pueden ser demostradas? "NEBP" se encarga de establecer que "la propia naturaleza del software hace improbable que alguna vez existan algunas balas de plata." Turski, en su excelente artículo de respuesta en el Congreso del IFIP lo expresó elocuentemente:

> *De todos los esfuerzos científicos equivocados, ninguno es más patético que la búsqueda de la piedra filosofal, una supuesta substancia para cambiar un metal común en oro. El propósito supremo de la alquimia, perseguido ardientemente por generaciones de investigadores generosamente patrocinados por gobernantes seculares y espirituales, es un extracto puro de fantasías, de la suposición común de que las cosas son como ellos quieren que sean. Es un sentimiento muy humano. Se requiere mucho esfuerzo aceptar la existencia de problemas insolubles. El deseo de ver una salida, en contra de todas las probabilidades, incluso cuando se ha demostrado que no existe, es bastante fuerte. Y la mayoría de nosotros tiene mucha compasión por esas almas valientes que tratan de conseguir lo imposible. Y así continúa. Se escriben disertaciones acerca de la cuadratura del círculo. Se preparan lociones para restaurar la pérdida de cabello y se venden bien. Se traman métodos para mejorar la productividad del software y se venden muy bien. Con mucha frecuencia todos estamos inclinados a seguir nuestro propio optimismo (o explotar los deseos optimistas de nuestros patrocinadores). Con mucha frecuencia todos nosotros estamos dispuestos a*

> *rechazar la voz de la razón y escuchar los cantos de sirena de los vendedores de panaceas.*[11]

Turski y yo hemos insistido en que los castillos en el aire *inhiben el progreso a largo plazo y malgastan el esfuerzo.*

Temas "pesimistas". Harel percibe que el pesimismo en "NEBP" surge de tres temas

- Una separación tajante entre esencia y accidente.
- Un tratamiento aislado de cada candidato a bala de plata.
- Predecir sólo por 10 años, en lugar de un tiempo suficiente en el cual "esperar una mejora significativa."

En cuanto al primero, ese es todo el argumento del artículo. Todavía creo que esta separación es absolutamente importante para entender por qué el software es difícil. Es una guía segura en cuanto al tipo de ataques que se deben realizar.

En cuanto al tratamiento aislado de los candidatos a balas de plata, en efecto "NEBP" así lo hace. Los distintos candidatos han sido propuestos uno por uno, con demandas extravagantes por cada uno *por separado.* Es justo evaluarlos uno por uno. No me opongo a las técnicas, sino esperar que ellas hagan milagros. Glass, Vessey y Conger en su artículo de 1992 ofrecen una prueba más que suficiente de la búsqueda vana por una bala de plata que aún no ha terminado.[12]

En cuanto a escoger 10 en contra de 40 años como un periodo de predicción, el periodo más corto fue en parte una concesión de que nuestros poderes predictivos nunca han sido buenos más allá de una década. ¿Quién de nosotros en 1975 predijo la revolución de las microcomputadoras de los 80's?

Hay otros motivos para el límite de una década: todas las exigencias hechas a los candidatos a balas de plata han tenido cierto apremio sobre ellos. No recuerdo a nadie que haya dicho "Invierta en mi panacea, y empezará ganando después de 10 años." Además, las proporciones rendimiento/precio del hardware se han incrementado quizás cien veces por década, y la comparación, aunque bastante inválida, es subconscientemente inevitable. Seguramente haremos progresos substanciales en los siguientes 40 años; un orden de magnitud por 40 años es apenas milagroso.

Los experimentos mentales de Harel. Harel propone un experimento mental en el cual postula como si "NEBP" hubiese sido escrito en 1952, en lugar de 1986, pero sosteniendo las mismas propuestas. Esto es como un *reducto add absurdum* para argumentar en contra del intento de separar la esencia del accidente.

El argumento no funciona. Primero, "NEBP" empieza afirmando que las dificultades accidentales dominaban en extremo las dificultades esenciales de la programación en la década de 1950, cosa que ya no hacen más, y que eliminándolas han logrado mejoras de órdenes de magnitud. Traducir ese argumento 40 años atrás no es razonable; en 1952 uno apenas podía imaginar afirmar que las dificultades accidentales no ocasionaban una gran parte del trabajo.

Segundo, el estado de cosas que Harel imagina que prevaleció en la década de los 50 es inexacto:

> *Ese era el tiempo en que en lugar de lidiar con el diseño de sistemas grandes y complejos, los programadores estaban en el negocio de desarrollar programas convencionales de una sola persona (que estaban en el orden de 100 a 200 líneas en un lenguaje de programación moderno) que eran para realizar tareas algorítmicas limitadas. Dada la tecnología y metodología disponibles entonces, tales tareas fueron igualmente formidables. Los fracasos, errores y plazos fallidos estaban por todos lados.*

Luego describe cómo los fracasos, errores y plazos fallidos considerados en los programas convencionales pequeños de una sola persona se mejoraron por un orden de magnitud en 25 años.

Pero el estado del arte en la década de los 50 no fue, de hecho, pequeños programas de una sola persona. En 1952, la Univac trabajó en el procesamiento del censo de 1950 con un programa complejo desarrollado por unos ocho programadores.[13] Otras máquinas estaban haciendo dinámica química, cálculos de difusión de neutrones, cálculos de desempeño de misiles, etc.[14] Ensambladores, enlazadores relocalizados y cargadores, sistemas interpretativos de punto flotante, etc. eran de uso común.[15] Para 1955 la gente estaba construyendo programas de negocios de 10 a 100 hombres-año.[16] Para 1956 la

General Electric tenía en operación un sistema de nómina en su planta de electrodomésticos de Louisville con un progama de más de 80,00 palabras. Para 1957, la computadora de defensa aérea SAGE ANFSQ/7 ya estaba en funcionamiento por dos años, y un sistema de comunicaciones de 75,00 instrucciones, en tiempo real, bidireccional y a prueba de errores estaba en operación en 30 lugares.[17] Difícilmente se puede sostener que es la evolución de técnicas para programas de una sola persona lo que describen en su mayoría los esfuerzos de la ingeniería de software desde 1952.

Y AQUÍ ESTÁ. Harel continúa para ofrecer su propia bala de plata, una técnica de modelado llamada "The Vanilla Framework." El propio enfoque no está descrito con suficiente detalle para evaluarlo, pero la referencia se remite a un artículo, y a un informe técnico que aparecerá en forma de libro a su debido tiempo.[18] El modelado aborda la esencia, la elaboración apropiada y el depurado de conceptos, así que es posible que Vanilla Framework sea revolucionario. Lo deseo. Ken Brooks informa que le pareció una metodología útil cuando lo probó en una tarea real.

Invisibilidad. Harel sostiene firmemente que gran parte de la construcción conceptual del software es intrínsecamente topológica por naturaleza y estas relaciones tienen su contraparte natural en representaciones espaciales/gráficas:

> *Usar un apropiado formalismo visual puede tener un efecto espectacular en los ingenieros y los programadores. Además, este efecto no está limitado a las cuestiones meramente accidentales; se ha encontrado que mejoró la calidad y prontitud de sus propios pensamientos. En el futuro el desarrollo de sistemas exitosos girará alrededor de las representaciones visuales. Primero conceptualizaremos, usando las entidades y las relaciones "apropiadas", y luego formularemos y reformularemos nuestras concepciones como una serie de modelos cada vez más amplios representados en una combinación apropiada de lenguajes visuales. Debe ser una combinación, puesto que los modelos del sistema tienen varias facetas, cada una de las cuales evoca diferentes tipos de imágenes mentales.*
>
> *. . . Algunos aspectos del proceso de modelado no han estado tan disponibles*

> *como otros en prestarse a una buena visualización. Por ejemplo, las operaciones algorítmicas sobre variables y estructuras de datos probablemente permanecerán textuales.*

Harel y yo somos muy parecidos. Lo que argumenté es que la estructura del software no está embebida en tres dimensiones, por lo que no existe un mapeo natural simple que vaya del diseño conceptual a un diagrama, ya sea en dos o más dimensiones. Él reconoce, y está de acuerdo, que se necesitan múltiples diagramas, cada uno cubriendo un aspecto distinto, y que otros aspectos no se representan bien en un diagrama en absoluto.

Comparto completamente su entusiasmo por el uso de diagramas como auxiliares del pensamiento y el diseño. Por mucho tiempo he disfrutado preguntar a candidatos a programadores, "¿Dónde está el próximo Noviembre?" Si la pregunta es muy críptica, entonces, "Dígame acerca de su modelo mental del calendario." Los programadores realmente buenos tienen sentidos espaciales sólidos; generalmente tienen modelos geométricos del tiempo; y con mucha frecuencia entienden la primera pregunta sin ninguna explicación. Tienen modelos altamente personalizados.

El Argumento de Jones – La Productividad Sigue a la Calidad

Los escritos de Capers Jones, primero en una serie de memorandos y más tarde en libro, ofrece una visión penetrante, que ha sido constatada por varios de mis corresponsales. "NEBP," como la mayoría de los escritos de ese tiempo, se enfocó en la *productividad,* el software de salida por unidad de entrada. Jones dice, "No. Mejor enfocarse en la *calidad*, y la productividad la seguirá."[19] Sostiene que los proyectos costosos y retrasados invierten la mayor parte del trabajo y tiempo extra buscando y reparando errores en las especificaciones, en el diseño y en la implementación. Ofrece datos que muestran una fuerte correlación entre la carencia de controles de calidad sistemáticos y desastres en el calendario. Lo creo. Boehm señala que la productividad cae de nuevo en tanto se persiga una calidad extrema, como en el software del transbordador espacial de IBM.

De forma similar, Coqui sostiene que los controles sistemáticos del desarrollo de software fueron elaborados en respuesta a consideraciones respecto

a la calidad (especialmente la elusión de desastres mayores) en lugar de consideraciones relacionadas con la productividad.

> *Pero observe: el objetivo de la aplicación de principios de Ingeniería a la producción de Software en los 70's fue mejorar la Calidad, las Pruebas, la Estabilidad y la Predictibilidad de los productos de software – no necesariamente la eficiencia en la producción de Software.*
>
> *La fuerza motriz para usar los principios de la Ingeniería de Software en la producción de software fue el temor a accidentes mayores que pudieran ser causados por tener artistas incontrolables responsables del desarrollo de sistemas cada vez más complejos.*[20]

Entonces ¿Qué Ha Pasado con la Productividad?

Cifras de productividad. Estas cifras son muy difíciles de definir, difíciles de calibrar y difíciles de encontrar. Capers Jones cree que para dos programas equivalentes en COBOL escritos con 10 años de diferencia, uno sin una metodología estructurada y otro con ella, la mejora es un factor de tres.

Ed Yourdon dice, "Veo gente obteniendo mejoras de cinco veces debido a las estaciones de trabajo y a las herramientas de software." Tom DeMarco cree "La expectativa de un incremento de un orden de magnitud en 10 años, debido a una canasta completa de técnicas fue optimista. Aún no he visto organizaciones logrando mejoras de un orden de magnitud."

Software en envoltorio de plástico – Compre; no construya. La evaluación de 1986 en "NEBP" ha probado ser correcta, al menos eso creo: "El desarrollo del mercado masivo es . . . la tendencia más profunda en el largo plazo en la ingeniería de software." Desde el punto de vista de la disciplina, el mercado masivo del software es casi una nueva industria comparada con el desarrollo del software a la medida, hecha en casa o fuera de ella. Cuando los paquetes se venden por millones – o incluso por miles – la calidad, la puntualidad, el rendimiento del producto y el costo del soporte llegan a ser cuestiones dominantes, en lugar del costo de desarrollo que es tan crucial en los sistemas a la medida.

Herramientas poderosas para la mente. La forma más espectacular de mejorar la productividad de los programadores de sistemas de gestión de información (SGI) es acudir a su tienda local de computadoras y comprar del anaquel lo que hubiesen construido. Esto no es ridículo; la disponibilidad de software en envoltorio de plástico - software comercial - es barato, y poderoso y ha satisfecho muchas necesidades que anteriormente hubieran ocasionado paquetes a la medida. Estas herramientas poderosas para la mente son más parecidas a taladros eléctricos, sierras y lijadoras que a grandes herramientas de producción compleja. La integración de estos en conjuntos compatibles y vinculados de forma cruzada tales como Microsoft Works y la mejor integrada ClarisWorks dan una flexibilidad inmensa. Y así como la colección de herramientas manuales eléctricas del dueño de casa, el uso frecuente de un conjunto pequeño, para muchas tareas diferentes, fomenta la familiaridad. Tales herramientas deben destacar la facilidad de uso para el usuario ocasional, no para el experto. Ivan Selin, presidente de American Managemente Systems, Inc., en 1987 me escribió:

> *Objeto su declaración de que los paquetes realmente no han cambiado mucho . . . : Creo que usted lanza muy a la ligera las principales implicaciones de su observación de que, [los paquetes de software] "pueden ser un tanto más generalizados y algo más individualizados que antes, pero no mucho." Incluso aceptando esta declaración literalmente, creo que los usuarios ven que los paquetes son más generalizados y más fáciles de personalizar, y que esta percepción conduce a los usuarios a estar mucho más receptivos a los paquetes. En la mayoría de los casos que mi compañía encuentra, es que los usuarios [finales], no la gente de software, son reacios a usar paquetes porque piensan que perderán características o funciones esenciales, y por lo tanto, la perspectiva de una fácil personalización es un argumento importante de venta para ellos.*

Creo que Selin tiene mucha razón – he subestimado tanto el grado de personalización de los paquetes como su importancia.

La Programación Orientada a Objetos – ¿Lo Hará una Bala de Latón?

Construir con piezas mayores. La ilustración que abre este capítulo nos recuerda que si uno ensambla un conjunto de piezas, cada una de ellas puede

ser compleja, y todas ellas fueron diseñadas para tener interfaces suaves y estructuras bastante ricas armonizarán rápidamente.

Un punto de vista de la programación orientada a objetos es que es una disciplina que impone *modularidad* e interfaces limpias. Un segundo punto de vista enfatiza el *encapsulamiento*, el hecho de que no se pueda observar, mucho menos diseñar, la estructura interna de las piezas. Otro punto de vista enfatiza la *herencia*, con su estructura *jerárquica* concomitante de clases, con funciones virtuales. Aún otro punto de vista enfatiza la *abstracción sólida del tipo de datos*, con su garantía de que un tipo de datos particular será manipulado sólo a través de operaciones propias.

Ahora podemos obtener cualquiera de estas disciplinas sin necesidad de tomar todo el paquete de Smalltalk o de C++ – muchas de ellas anteriores a la tecnología orientada a objetos. El atractivo del enfoque orientado a objetos es el de una píldora multivitamínica: de un tirón (es decir, el reentrenamiento del programador), se obtiene todas ellas. Es un concepto muy prometedor.

¿Por qué la técnica orientada a objetos ha crecido lentamente? En los nueve siguientes nueves años a partir de "NEBP," las expectativas han crecido constantemente. ¿Por qué su crecimiento ha sido lento? Las teorías abundan. James Coggins, autor por cuatro años de la columna, "Lo Mejor de `comp.lang.c++`" en *The C++ Report*, ofrece esta explicación:

> *El problema es que los programadores en O-O han estado experimentando en aplicaciones incestuosas y enfocándose en una abstracción baja, en lugar de una alta. Por ejemplo, han estado construyendo clases tales como* listas enlazadas *o* conjuntos *en lugar de clases tales como interfaz de usuario o* haces de radiación *o* modelos de elementos finitos. *Desafortunadamente la misma y poderosa verificación de tipos en C++ que ayuda a los programadores a evitar errores también dificulta la construcción de cosas grandes a partir de pequeñas.*[21]

Él regresa al problema fundamental del software, y sostiene que una manera de abordar las necesidades del software no satisfechas es incrementar el tamaño de la fuerza laboral inteligente mediante la capacitación y la incorporación de nuestros clientes. Este es un argumento para el diseño descendente:

> *Si diseñamos clases de grano grueso que aborden conceptos con los que nuestros clientes ya están trabajando, ellos podrán entender y cuestionar el diseño mientras se desarrolla, y podrán cooperar en el diseño de los casos de prueba. Mis colaboradores de oftalmología no se preocupan acerca de las pilas; sí se preocupan por las descripciones de los polinomios de Legendre de la forma de las córneas. Pequeños encapsulamientos producirán pequeños beneficios.*

David Parnas, cuyo artículo fue uno de los que dieron origen a los conceptos orientados a objetos, observa el asunto de forma muy diferente. Me escribe:

> *La respuesta es simple. Es porque [O-O] ha estado ligado a una variedad de lenguajes complejos. En lugar de enseñar a la gente que O-O es un tipo de diseño, y dándoles principios de diseño, la gente ha enseñado que O-O es el uso de una herramienta especial. Podemos escribir buenos o malos programas con cualquier herramienta. A menos que enseñemos a la gente cómo diseñar, el lenguaje importa muy poco. El resultado es que la gente hace malos diseños con estos lenguajes y obtiene muy poco valor de ellos. Si el valor es pequeño, no lo captarán.*

Costos anticipados, beneficios futuros Mi propia creencia es que las técnicas orientadas a objetos tienen un caso peculiarmente severo de una enfermedad que caracteriza a muchos adelantos metodológicos. Los costos directos son muy substanciales – ante todo el reentrenamiento de los programadores para pensar de una forma nueva, pero también la inversión extra para confeccionar funcionalidades dentro de clases generalizadas. Los beneficios, que pienso que son reales y no meramente supuestos, ocurren a lo largo del ciclo de desarrollo; aunque los grandes beneficios rendirán frutos durante las actividades sucesivas de construcción, extensión y mantenimiento. Cogging dice, "las técnicas Orientadas a Objetos no harán que el desarrollo del primer proyecto o el siguiente sean más rápidos. El quinto de esa familia irá a toda velocidad"[22]

Apostar cantidades significativas de dinero anticipadamente en aras de beneficios futuros pero inciertos es lo que los inversionistas hacen todos los días. Sin embargo, en muchas organizaciones de programación se requiere de una auténtica valentía de gestión, una mercancía más escasa que la competencia técnica o la destreza administrativa. Creo que el grado extremo de

los costos anticipados y beneficios diferidos son el principal factor sin excepción de la lenta adopción de las técnicas O-O. Aún así, C++ parece estar reemplazando contínuamente a C en mucha comunidades.

¿Qué Ocurre con el Reuso?

La mejor manera de atacar la esencia de la construcción del software es no construirlo en absoluto. El paquete de software es sólo una de muchas formas de hacer esto. El reuso de un programa es otro. Asimismo, la promesa de un fácil reuso de clases, con una fácil personalización a través de la herencia, es uno de los atractivos más fuertes de las técnicas orientadas a objetos.

Como generalmente es el caso, a medida que se adquiere cierta experiencia con una nueva manera de hacer negocios la nueva forma no es tan simple como parece al principio.

Los programadores, obviamente, siempre han reusado su propio trabajo. Jones dice,

> *La mayoría de los programadores experimentados tienen bibliotecas privadas que les permiten desarrollar software con cerca del 30% de código reusado del volumen total. La reusabilidad en los niveles corporativos apunta al 75% de código reusado del total, y requiere bibliotecas especializadas y apoyo administrativo. El código reusable corporativo también implica cambios en la contabilidad del proyecto y en las prácticas de medición para así acreditar la reusabilidad.*[23]

W. Huang propuso organizar fábricas de software con una gestión matricial de especialistas prácticos, para aprovechar la propensión natural de todos a reusar su propio código.[24]

Van Snyder de JPL me señala que la comunidad de software matemático tiene una larga tradición de reuso de software:

> *Conjeturamos que las barreras para reusar no están del lado del productor, sino del lado del cliente. Si un ingeniero de software, un cliente potencial de componentes de software estandarizados, percibe que es más costoso encontrar un componente que cumpla sus necesidades, y que además es más costoso verificarlo, que escribir uno nuevo, se escribirá un nuevo componentes duplicado. Observe que arriba dijimos percibe. Y, no importa cuál sea el verdadero costo*

de reconstrucción.
El reuso ha sido exitoso en el software matemático por dos razones: (1) Es arcano, requiere una enorme aportación intelectual por línea de código, y (2) existe una nomenclatura rica y estándar, llamémosle matemática, para describir la funcionalidad de los componentes. Así que el costo para reconstruir un componente matemático de software es alto, y el costo por descubrir la funcionalidad de un componente existente es bajo. La larga tradición de revistas profesionales que publican y recopilan algoritmos, y los ofrecen a precios modestos, y en cuestiones comerciales ofrecen algoritmos de muy alta calidad a un costo más alto pero todavía modesto, hace que descubrir un componente que cumpla con nuestras necesidades sea más simple que en muchas otras disciplinas, donde a veces no es posible especificar las necesidades de forma precisa y concisa . Estos factores abonan para hacer más atractivo reusar en vez de reinventar el software matemático.

El mismo fenómeno de reuso se halla entre varias comunidades, como las que construyen código para reactores nucleares, modelos climáticos, modelos oceánicos, y por las mismas razones. Las comunidades crecen con los mismos libros de texto y notaciones estándar.

¿Cómo le va hoy en día al reuso a nivel corporativo? Muchos estudios; relativamente poca práctica en los Estados Unidos; informes anecdóticos de mayor reuso en el extranjero.[25]

Jones ha reportado que todos los clientes de su firma con más de 5000 programadores tienen una investigación formal del reuso, mientras que menos de 10 por ciento de los clientes con menos de 500 programadores lo hacen.[26] Él afirma que en las industrias con el mayor potencial de reuso, la investigación sobre la reusabilidad (no el uso) "es activa y enérgica, incluso si no es todavía totalmente exitosa." Ed Yourdon informa de una casa de software en Manila que tiene a 50 de sus 200 programadores construyendo sólo módulos reusables para que el resto los utilice, "He visto unos cuantos casos – la adopción se debe a factores *organizativos* como la estructura de recompensa, no a factores técnicos."

DeMarco me dice que la disponibilidad de un mercado masivo de paquetes y su idoneidad como proveedor de funcionalidades genéricas como los sistemas de bases de datos ha reducido substancialmente tanto la presión como

la utilidad marginal de reusar módulos propios del código de aplicación. "De cualquier manera los módulos reusables son propensos a ser funcionalidades genéricas."
Parnas escribe,

> *Reusar es algo mucho más fácil de decir que de hacer. Hacerlo requiere por un lado un buen diseño y por otro una buena documentación. Incluso cuando vemos un buen diseño, lo cual es todavía infrecuente, no veremos los componentes reusados sin una buena documentación.*

Ken Brooks comenta sobre la dificultad de anticipar *qué* generalización demostrará ser necesaria: "Sigo forzando las cosas incluso en el quinto uso de mi propia biblioteca personal de interfaz de usuario."

El reuso real parece ser sólo el inicio. Jones informa que se ofrecen algunos módulos de código reusable en el mercado abierto a precios entre el 1 y 20 por ciento del costo de desarrollo normal.[27] DeMarco dice,

> *Me estoy desanimando acerca de todo el fenómeno del reuso. Existe una casi total ausencia de un teorema de existencia para el reuso. El tiempo ha confirmado que existe un gran gasto en hacer que las cosas sean reusables.*

Yourdon estima este gran gasto. "Una buena regla empírica es que tales componentes reusables requerirán el doble de esfuerzo que un componente de 'un solo tiro'."[28] Considero que ese gasto es exactamente el esfuerzo de producir el componente, discutido en el Capítulo 1. Así que mi estimación de la razón de esfuerzo sería el triple.

Claramente estamos viendo muchas formas y variedades de reuso, pero no tanto como habíamos esperado por ahora. Aún nos queda mucho por aprender.

Aprender Vocabularios Grandes – Un Problema Predecible pero no predicho para el Reuso del Software

A medida que pensamos a un nivel más alto, son más numerosos los elementos del pensamiento primitivo con los que uno trata. Por lo tanto, los lenguajes de programación son mucho más complejos que los lenguajes de máquina, y los lenguajes naturales son todavía más complejos. Los lenguajes de más

alto nivel tienen vocabularios más grandes, una sintaxis más compleja y una semántica más rica.

Como disciplina, no hemos reflexionado sobre las implicaciones de este hecho para el reuso de programas. Para mejorar la calidad y la productividad, queremos construir programas componiendo fragmentos de funciones depuradas que están substancialmente más elevadas que las declaraciones en los lenguajes de programación. Por lo tanto, ya sea que hagamos esto a través de bibliotecas de clases o bibliotecas de procedimientos, debemos encarar el hecho de que estamos aumentando radicalmente el tamaño de nuestro vocabulario de programación. El aprendizaje del vocabulario no constituye una parte pequeña de la barrera intelectual hacia el reuso.

Así actualmente la gente tiene bibliotecas de clases con algo más de 3,000 miembros. Muchos objetos requieren la especificación de 10 a 20 parámetros y variables opcionales. Cualquiera que programe con esa biblioteca debe, aprender la sintaxis (las interfaces externas) y la semántica (el comportamiento funcional detallado) de sus miembros si van a llevar a cabo todo el reuso potencial.

Esta tarea está lejos de ser inútil. Los hablantes nativos normalmente usan un vocabulario de más de 10,000 palabras, la gente educada muchas más. De alguna manera aprendemos la sintaxis y la muy sutil semántica. Diferenciamos de forma correcta entre *gigante, grande, vasto, enorme, colosal;* la gente no sólo habla de desiertos colosales o elefantes vastos.

Necesitamos investigar para adecuar al problema del reuso de software el gran cuerpo de conocimiento así como de qué manera la gente adquiere el lenguaje. Algunas de las lecciones son inmediatamente obvias:

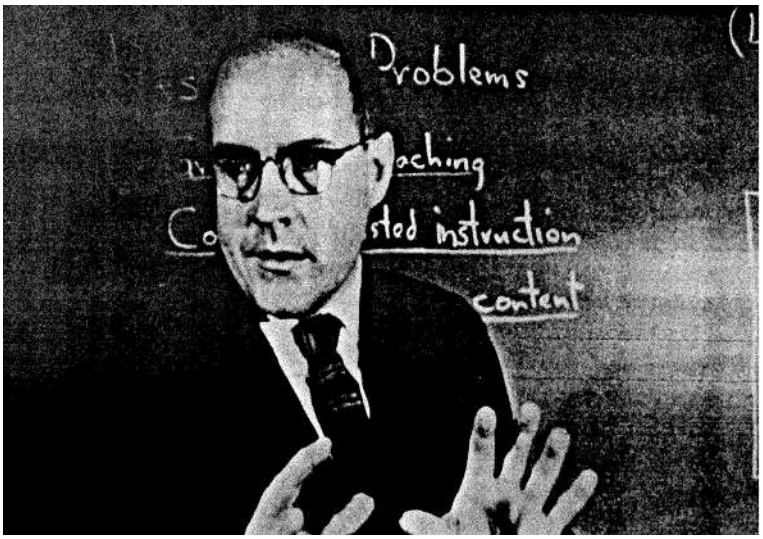
- La gente aprende en contextos de oraciones, así pues, necesitamos publicar muchos ejemplos de productos integrados, no sólo bibliotecas de partes.
- La gente no memoriza nada, más bien deletrea. Aprenden la sintaxis y la semántica incrementalmente, en contexto, a través del uso.
- La gente agrupa las reglas de composición de palabras a través de clases sintácticas, no a través de subconjuntos compatibles de objetos.

Una Red sobre las Balas – Una Postura Sin Cambio

Así pues, regresamos a los fundamentos. La complejidad *es* el asunto en el que estamos involucrados, y la complejidad es la que nos limita. R.L Glass en un escrito de 1988 resumía exactamente mis puntos de vista de 1995:

> *En retrospectiva, y ¿Qué tienen que decirnos Parnas y Brooks? Que el desarrollo del software es un asunto del pensamiento conceptual. Que las soluciones mágicas no están a la vuelta de la esquina. Que es el momento para que sus practicantes examinen mejoras evolutivas en lugar de esperar – o desear – mejoras revolucionarias.*
>
> *Algunos en el campo del software consideran que esta es una imagen desalentadora. Son de los que todavía pensaban que los avances estaban al alcance de la mano.*
>
> *Pero algunos de nosotros – aquellos lo suficientemente quisquillosos para pensar que somos realistas – vemos esto como un aliento de aire fresco. Por último, podemos concentrarnos en algo un poco más viable que los castillos en el aire. Ahora, tal vez, podemos continuar con las mejoras incrementales a la productividad del software que son posibles, en lugar de esperar por los grandes avances que probablemente nunca lleguen.*[29]

18
Las Propuestas de El Mítico Hombre-Mes: Verdaderas o Falsas?


Problems
content

18
Las Propuestas de El Mítico Hombre-Mes: Verdaderas o Falsas?

La brevedad es muy buena,
Ya sea que nos entiendan, o no.

SAMUEL BUTLER, Hudibras

Brooks sostiene una propuesta, 1967
Foto de J. Alex Langley para la revista *Fortune*

Hoy sabemos mucho más acerca de la ingeniería de software de lo que se sabíamos en 1975. ¿Qué afirmaciones de la edición original de 1975 han sido respaldadas por los datos y la experiencia? ¿Cuáles han sido refutadas? ¿Cuáles se han vuelto obsoletas por este mundo en constante cambio? Para ayudarle a juzgar, aquí está en una forma de resumen la esencia del libro de 1975 – las afirmaciones que yo creía eran ciertas: hechos y generalizaciones tomadas de la experiencia – extraídas sin cambio de sentido. (Usted podría preguntar, "si esto es todo lo que dijo el libro original, ¿por qué tomó 177 páginas para decirlo?") Los comentarios entrecomillados son nuevos.

La mayor parte de estas propuestas son operacionalmente demostrables. Mi deseo al publicarlas en forma austera es para concentrar los pensamientos, las evaluaciones y los comentarios del lector.

Capítulo 1. El Pozo de Brea

1.1 Un producto de sistemas de programación requiere nueve veces más de esfuerzo que los programas componente escritos de forma separada para un uso privado. Estimo que producirlos impone un factor de tres; y que diseñar, integrar y probar los componentes en un sistema coherente impone otro factor de tres, y que el costo de estos componentes son esencialmente independientes entre sí.

1.2 El oficio de la programación "gratifica el anhelo creativo inmerso profundamente en nosotros y deleita las sensibilidades que tenemos en común con todas las personas," y proporciona cinco tipos de satisfacciones:

- La satisfacción de hacer cosas
- La satisfacción de hacer cosas que sean útiles para otras personas
- La fascinación de construir cosas tipo rompezabezas de piezas móviles interconectadas
- La satisfacción del continuo aprendizaje, de una tarea no repetitiva
- La satisfacción de trabajar con un medio tan tratable – un instrumento puramente intelectual – que sin embargo, existe, se mueve y funciona en una forma en que los objetos de puras palabras no lo hacen.

1.3 Asimismo el oficio tiene infortunios intrínsecos.

- Ajustarse al requisito de perfección es la parte más difícil de aprender a programar.
- Otros asignan nuestros objetivos y uno depende de cosas (especialmente programas) que no puede controlar; la autoridad no se equipara a la responsabilidad.
- Esto suena peor de lo que es: la autoridad real proviene del impulso del cumplimiento.
- Con cualquier actividad creativa vienen pesadas horas de meticuloso trabajo; la programación no es la excepción.
- El proyecto de programación converge más lentamente cuanto más se acerca al final, mientras uno espera que converja más rápido a medida que se aproxima al final.
- El producto está siempre amenazado por la obsolescencia antes de su finalización. Un tigre de verdad jamás se equipara a uno de papel, a menos que queramos usarlo de verdad.

Capítulo 2. El Mítico Hombre-Mes

2.1 Gran parte de los proyectos de programación han fracasado más por la escasez de tiempo de calendario que debido a otras causas combinadas.

2.2 La buena cocina toma tiempo; algunas tareas no pueden apresurarse sin estropear el resultado.

2.3 Todo los programadores son optimistas: "Todo saldrá bien."

2.4 Debido a que los programadores construyen a partir de un instrumento puramente intelectual, esperamos pocas dificultades en la implementación.

2.5 Pero nuestras *ideas* en sí mismas son defectuosas, así que tenemos errores.

2.6 Nuestras técnicas de estimación, basadas en la contabilidad de costos, confunden esfuerzo con progreso. *El hombre-mes es un mito falaz y peligroso, porque implica que los hombres y los meses son intercambiables.*

2.7 Dividir una tarea entre varias personas ocasiona un trabajo extra de comunicación, esto es, entrenamiento e intercomunicación.

2.8 Mi regla empírica es 1/3 del calendario para el diseño, 1/6 para la codificación, 1/4 para la prueba de componentes, y 1/4 para la prueba del sistema.

2.9 Como disciplina, carecemos de datos de estimación.
2.10 Debido a que tenemos incertidumbre acerca de nuestras estimaciones del calendario, carecemos del valor para defenderlas tenazmente en contra de las presiones de los gestores y clientes.
2.11 La ley de Brooks: Añadir mano de obra a un proyecto de software retrasado lo retrasa aún más.
2.12 Añadir gente a un proyecto de software aumenta el esfuerzo total que se necesita de tres formas: por el trabajo y la interrupción del propio reparto, por el entrenamiento del nuevo personal y la intercomunicación añadida.

Capítulo 3. El Equipo Quirúrgico

3.1 Los programadores profesionales muy buenos son *diez veces* más productivos que los malos, con el mismo nivel de entrenamiento y dos años de experiencia. (Sackman, Grant, y Erickson)
3.2 Los datos de Sackman, Grant, y Erickson no mostraron ninguna correlación en absoluto entre experiencia y rendimiento. Dudo de la universalidad de ese resultado.
3.3 Un equipo pequeño e inteligente es mejor – tan pocas mentes como sea posible.
3.4 Un equipo de dos, con un líder, es a menudo el mejor uso que se le puede dar a las mentes. [Observe el plan de dios para el matrimonio.]
3.5 Un equipo pequeño e inteligente es muy lento para sistemas realmente grandes.
3.6 Gran parte de las experiencias con sistemas realmente grandes muestran que el enfoque de la fuerza bruta para el proceso de ampliación es costoso, lento, ineficiente y produce sistemas que no están integrados conceptualmente.
3.7 Un programador principal, la organización del equipo quirúrgico ofrece una manera de obtener la integridad del producto a partir de unas cuantas mentes y la productividad total de muchos ayudantes, con una comunicación radicalmente reducida.

Capítulo 4. Aristocracia, Democracia, y Diseño de Sistemas

4.1 "La integridad conceptual es *la* consideración *más* importante en el diseño de sistemas."
4.2 "La proporción entre funcionalidad y complejidad conceptual es la prueba final del diseño de sistemas," no sólo la riqueza de la funcionalidad. [Esta proporción es una medida de la facilidad de uso, válida a través del uso simple como complicado.]
4.3 Para lograr la integridad conceptual, un diseño debe proceder de una mente o de un pequeño número de mentes en sintonía.
4.4 "Separar el esfuerzo arquitectónico de la implementación es una forma muy poderosa de obtener la integración conceptual en proyectos muy grandes." [También en los pequeños.]
4.5 "Si un sistema implica tener integridad conceptual, alguien debe controlar esos conceptos. Esa es una aristocracia que no necesita excusa."
4.6 La disciplina es buena para el arte. El suministro externo de una arquitectura mejora, no estorba, el estilo creativo del grupo de implementadores.
4.7 Un sistema integrado conceptualmente es más rapido de construir y probar.
4.8 Gran parte de la arquitectura, la implementación y la realización del software puede avanzar en paralelo. [Asimismo el diseño del hardware y software pueden avanzar en paralelo.]

Capítulo 5. El Efecto del Segundo Sistema

5.1 Una comunicación pronta y continua puede brindar al arquitecto buenos indicios del costo y al constructor confianza en el diseño, sin disipar la clara división de responsabilidades.
5.2 Cómo puede un arquitecto influir en la implementación exitosamente.
- Recordar que el constructor tiene la responsabilidad creativa de la implementación, el arquitecto solo sugiere.
- Estar siempre preparado para sugerir una manera de implementar cualquier cosa que se especifique; estar preparado para aceptar a su vez cualquier otra manera de hacerlo bien.
- Tratar de forma tranquila y privada tales sugerencias.

- Estar listo a renunciar al crédito por las mejoras sugeridas.
- Escuche sugerencias del constructor por mejoras en la arquitectura.

5.3 El segundo sistema es el más peligroso que una persona jamás diseña; la tendencia general es a sobrediseñarlo.

5.4 El OS/360 es un buen ejemplo del efecto del segundo sistema. [Windows NT parece ser un buen ejemplo de los 90's.]

5.5 Una practica valiosa es asignar a las funciones valores *a priori* en bytes y microsegundos.

Capítulo 6. Pasar la Voz

6.1 Incluso cuando un equipo de diseño sea grande, la redacción de los resultados debe ser condensada por una o dos personas, para que las minidecisiones sean consistentes.

6.2 Es importante definir explícitamente las partes de una arquitectura que *no* están prescritas tan cuidadosamente como las que sí lo están.

6.3 Se necesita por un lado una definición formal de un diseño, por su precisión, y por otro una definición en prosa por su coherencia.

6.4 Una de las dos definiciones, la formal o en prosa, debe ser la estándar y la otra derivada. Cualquiera de las dos puede servir en cualquiera de los roles.

6.5 Una implementación, incluyendo una simulación, puede servir como una definición arquitectónica; este uso tiene tremendas desventajas.

6.6 La incorporación directa es una técnica muy limpia para imponer un estándar arquitectónico en software. [En hardware, también – considere la interfaz VIMP de la Mac construida dentro de la ROM.]

6.7 Una definición arquitectónica "será mas limpia y la disciplina [arquitectónica] más rigurosa si al menos inicialmente se construyen dos implementaciones."

6.8 Es importante permitir las interpretaciones telefónicas de un arquitecto en respuesta a las dudas de los implementadores; es imperativo registrarlas y publicarlas. [Hoy en día el correo electrónico es el medio idóneo.]

6.9 "El mejor amigo del gestor de proyecto es su adversario diario, la organización de un probador del producto independiente."

Capítulo 7. Por Qué Fracasó la Torre de Babel?

7.1 El proyecto de la Torre de Babel fracasó por la falta de *comunicación* y su consecuencia, *la organización.*

Comunicación

7.2 "El desastre del calendario, el desajuste funcional y los errores del sistema todos surgen porque la mano izquierda no sabe lo que hace la derecha." Los equipos se alejan poco a poco basados en conjeturas.

7.3 Los equipos deberían comunicarse entre sí de tantas formas como sea posible: informalmente, a través de reuniones habituales del proyecto, con resúmenes técnicos, y a través de un libro de trabajo formal compartido del proyecto. [Y por correo electrónico.]

Libro de Trabajo del Proyecto

7.4 Un libro de trabajo "no es tanto un documento separado sino una estructura impuesta a los documentos que el proyecto producirá de todos modos."

7.5 "*Todos* los documentos del proyecto deben ser parte de esta estructura [libro de trabajo]"

7.6 La estructura del libro de trabajo necesita diseñarse *meticulosamente* y con *antelación.*

7.7 Estructurando adecuadamente desde un principo la documentación en curso "moldea la escritura posterior en segmentos que se ajustan a esa estructura" y mejorará los manuales del producto.

7.8 "*Todo* miembro del equipo debería ver *todo* el material [libro de trabajo]." [Ahora diría, que todo miembro del equipo *debería ser capaz* de verlo todo. Para eso, bastaría con páginas Web.]

7.9 La actualización oportuna es de suma importancia.

7.10 Se debe atraer la atención del usuario especialmente a los cambios realizados desde su última lectura, con observaciones acerca de su importancia.

7.11 El libro de trabajo del proyecto del OS/360 empezó en papel y se cambió a microfichas.

7.12 Actualmente [aun en 1975], el cuaderno electrónico compartido es un mecanismo mucho mejor, más barato y más simple para lograr todos

estos objetivos.

7.13 Aún se tiene que marcar el texto con barras de cambio [o su equivalente funcional] y fechas de revisión. Aún se necesita un resumen electrónico de cambios en forma de pila (LIFO).

7.14 Parnas sostiene firmemente de que el objetivo de que todos vean todo es *totalmente erróneo*; las piezas deberían ser encapsuladas tal que nadie necesite o se le permita ver el interior de cualquier pieza que no sea la suya, sólo deberían ver las interfaces.

7.15 La propuesta de Parnas es una receta para el desastre. [*Parnas me ha convencido de lo contrario, y he cambiado de opinión por completo.*]

Organización

7.16 El propósito de la organización es reducir la cantidad de comunicación y coordinación necesaria.

7.17 La organización encarna *la división del trabajo y la especialización de funciones* para evitar la comunicación.

7.18 La organización tradicional tipo árbol refleja el principio de estructura de *autoridad* de que nadie puede servir a dos amos.

7.19 La estructura de *comunicación* en una organización es una red, no un árbol, así que se tienen que idear todo tipo de mecanismos especiales de organización ("líneas punteadas") para superar las deficiencias de comunicación de la organización estructurada como árbol.

7.20 Todo proyecto secundario tiene dos roles de liderazgo a ser ocupados, el del *productor* y el del *director técnico* o *arquitecto.* Las funciones de los dos roles son bastante distintos y requieren cualidades diferentes.

7.21 Cualquiera de las tres relaciones entre los dos roles puede ser bastante eficaz.

- El productor y el director pueden ser la misma persona.
- El productor puede ser el jefe y el director su mano derecha.
- El director puede ser el jefe y el productor su mano derecha.

Capítulo 8. Predecir la Jugada

8.1 No se puede estimar exactamente el esfuerzo total o el calendario de un proyecto de programación simplemente estimando el tiempo de codificación y multiplicando las demás partes de la tarea por factores.

8.2 Los datos para construir pequeños sistemas aislados no son aplicables a proyectos de programación de sistemas.
8.3 La programación se incrementa como una potencia del tamaño del programa.
8.4 Algunos estudios publicados muestran que el exponente es de alrededor de 1.5. [*Los datos de Boehm no concuerdan en absoluto con esto, sino que varían de 1.05 a 1.2*][1]
8.5 Los datos de ICL de Portman muestran que los programadores de tiempo completo dedican sólo alrededor del 50 por ciento de su tiempo a programar y a depurar, en contra de otras tareas de tipo general.
8.6 Los datos de IBM de Aron muestran una productividad que varía desde 1.5 K líneas de código (KLDC) por hombre-año a 10 KLDC/hombre-año en función del número de interacciones entre las partes del sistema.
8.7 Los datos de Bell Labs de Harr muestran productividades en trabajos tipo sistemas operativos realizar alrededor de 10 KLDC/hombre-año y en tareas tipo compilador alrededor de 2.2 KLDC/hombre-año para productos terminados.
8.8 Los datos del OS/360 de Brooks están de acuerdo con los datos de Harr. 0.6-0.8 KLDC/hombre-año en sistemas operativos y 2-3 KLDC/hombre-año en compiladores.
8.9 Los datos del proyecto MULTICS del MIT de Corbató muestran una productividad de 1.2 KLDC/hombre-año en una mezcla de sistemas operativos y compiladores, aunque son líneas de código en PL/I, mientras todos los demás datos son líneas de código en ensamblador!
8.10 La productividad parece constante en términos de declaraciones elementales.
8.11 La productividad de la programación se puede incrementar casi cinco veces cuando se utiliza un lenguaje de alto nivel adecuado.

Capítulo 9. Diez Libras en un Saco de Cinco Libras

9.1 Aparte del tiempo de ejecución, el *espacio de memoria* ocupado por un programa representa un costo importante. Esto es especialmente cierto para los sistemas operativos, donde gran parte de ellos son programas residentes todo el tiempo.
9.2 Aun así, el dinero invertido en memoria para programas residentes pue-

de rendir muy buenos dividendos funcionales por dólar, mejor que otras formas de inversión en configuración. El tamaño del programa no es malo; el tamaño innecesario sí lo es.

9.3 El constructor de software debe fijar objetivos de tamaño, el tamaño del control, e idear técnicas de reducción de ese tamaño, así como los constructores de hardware hacen con los componentes.

9.4 Los presupuestos del tamaño deben ser explícitos no sólo acerca del tamaño del espacio de memoria residente sino también acerca de los accesos a disco producidos por las búsquedas del programa.

9.5 Los presupuestos del tamaño tiene que estar ligados a las asignaciones funcionales; definir exactamente qué debe hacer un módulo cuando se especifique cuán grande debe ser.

9.6 En equipos grandes, los subequipos tienden a suboptimizar por cumplir sus propios objetivos en lugar de pensar acerca del efecto total en el usuario. Esta falla en la orientación es un peligro importante en proyectos grandes.

9.7 Durante toda la implementación, los arquitectos del sistema deben mantener una vigilancia constante para asegurar una integridad perdurable del sistema.

9.8 Fomentar una actitud de sistema total y orientado al usuario bien puede ser la función más importante del gestor de programación.

9.9 Una decisión política temprana es decidir cuánto grano fino tendrá el usuario en la elección de opciones, puesto que empaquetándolos en grupos ahorra espacio de memoria [y generalmente costos de mercadotecnia].

9.10 Una decisión crucial es el tamaño del área temporal, por lo tanto, de la cantidad de programa por búsqueda en disco, puesto que el rendimiento es una función super lineal de ese tamaño. [Toda esta decisión se ha hecho obsoleta, primero debido a la memoria virtual, luego por la memoria física barata. Hoy en día los usuarios compran suficiente memoria física para almacenar todo el código de sus principales aplicaciones.]

9.11 Para lograr buenos compromisos de espacio-tiempo, un equipo debe estar entrenado en técnicas de programación propias de un lenguaje o máquina particular, especialmente si es nueva.

9.12 La programación tiene una tecnología, y todo proyecto necesita una

biblioteca de componentes estándar.

9.13 Las bibliotecas de programas deben tener dos versiones de cada componente, la rápida y la comprimida. [Esto parece ser obsoleto actualmente.]

9.14 Los programas ligeros, sobrios y rápidos son casi siempre el resultado de *avances estratégicos,* en lugar de tácticas inteligentes.

9.15 A menudo, este avance será un nuevo *algoritmo*.

9.16 Con mayor frecuencia, el adelanto vendrá de rehacer la *representación* de los datos o tablas. *La representación es la esencia de la programación.*

Capítulo 10. La Hipótesis Documental

10.1 "La hipótesis: Entre un montón de papeles, un pequeño número de documentos llega a ser el apoyo crítico alrededor del cual gira toda gestión de proyecto. Son las principales herramientas personales del gestor."

10.2 Para un proyecto de desarrollo de computadoras, los documentos críticos son: los objetivos, el manual, el calendario, el presupuesto, el organigrama, la asignación de espacio y la estimación, esto es, el pronóstico y el precio de la máquina misma.

10.3 Para el departamento de una universidad, los documentos críticos son semejantes: los objetivos, los requisitos de grado, la descripción del curso, las propuestas de investigación, el horario de clases, y el plan de estudios, el presupuesto, la asignación de espacio, y las asignaciones de personal y colaboradores graduados.

10.4 Para un proyecto de programación, las necesidades son las mismas: los objetivos, el manual del usuario, los documentos internos, el calendario, el presupuesto, el organigrama y la asignación de espacio.

10.5 Por lo tanto, aun en pequeños proyectos, el gestor debería formalizar desde un principio tal conjunto de documentos.

10.6 La preparación de cada documento de este pequeño conjunto centra las ideas y plasma la discusión. El acto de escribir requiere de centenares de mini-decisiones, y su existencia es lo que distingue las políticas claras y exactas de las difusas.

10.7 El mantenimiento de cada documento crítico brinda un mecanismo de supervisión y de alerta del estado.

10.8 Cada documento por sí mismo sirve como una lista de control y una

base de datos.

10.9 El trabajo primordial del gestor de proyecto es mantener a todos avanzando en la misma dirección.

10.10 La principal tarea diaria del gestor de proyecto es la comunicación, no la toma de decisiones: los documentos comunican los planes y las decisiones a todo el equipo.

10.11 Sólo una pequeña porción del tiempo del gestor de proyecto técnico – quizá el 20 por ciento – se invierte en tareas donde necesita información externa.

10.12 Por esta razón, el tan aclamado concepto de mercado de un "sistema de información total de gestión" para apoyar a ejecutivos no está basado en un modelo válido de comportamiento del ejecutivo.

Capítulo 11. Planifique Desechar

11.1 Los ingenieros químicos han aprendido a no tomar un proceso de la mesa de trabajo de laboratorio a la fábrica en un solo paso, sino a construir una *planta piloto* experimentando con cantidades a mayor escala y operando en ambientes no protegidos.

11.2 Este paso intermedio es igualmente necesario para los productos de programación, aunque los ingenieros de software todavía no realizan de forma rutinaria pruebas de campo del sistema piloto antes de emprender la liberación del producto real. [Esto ahora se ha convertido en una práctica común, con una versión beta. Esto no es lo mismo que un prototipo con una funcionalidad limitada, una versión alfa, al que también me inclinaría.]

11.3 Para la mayoría de los proyectos, el primer sistema construido es apenas utilizable: es muy lento, o muy grande, o muy difícil de usar, o los tres juntos.

11.4 El descartar y rediseñar se puede hacer en un solo bloque, o pieza por pieza, *de cualquier modo eso se hará.*

11.5 Liberar el primer sistema, el desechable, a los usuarios comprará tiempo, pero sólo a costa de la angustia del usuario, la distracción de los constructores que le dan soporte mientras realizan el rediseño, y una mala reputación para el producto que será difícil de superar.

11.6 Por lo tanto, *planifique desechar; lo hará de cualquier modo.*

11.7 "El programador entrega satisfacción a la necesidad de un usuario en vez de un producto tangible." (Cosgrove)

11.8 Conforme el programa se contruya, pruebe y use, la necesidad real y la percepción del usuario de esa necesidad *cambiarán*.

11.9 La maleabilidad como la invisibilidad del producto de software exponen (de forma notable) a sus constructores a cambios perpetuos en los requisitos.

11.10 Algunos cambios legítimos en los objetivos (y en las estrategias de desarrollo) son inevitables, y es mejor estar preparados para ellos que suponer que nunca llegarán.

11.11 Las técnicas de planificación de un producto de software para el cambio, en especial la programación estructurada con una documentación meticulosa de la interfaz de los módulos, son bien conocidas aunque no practicadas universalmente. También es útil, donde sea posible, el uso de técnicas manejadas por tablas. [El costo y el tamaño de las memorias actuales mejoran dichas técnicas cada vez más.]

11.12 Utilizar un lenguaje de alto nivel, operaciones en tiempo de compilación, incorporación de declaraciones por referencia y técnicas de autodocumentación para la reducción de errores inducidos por los cambios.

11.13 Cuantificar los cambios en versiones numeradas bien definidas. [Actualmente una práctica estándar.]

Planifique la Organización para el Cambio

11.14 La renuencia de los programadores a documentar los diseños no viene tanto de la pereza como de la vacilación por emprender la defensa de decisiones que el diseñador sabe que son provisionales. (Cosgrove)

11.15 La estructuración de una organización para el cambio es mucho más difícil que diseñar un sistema para el cambio.

11.16 El jefe del proyecto debe trabajar en mantener a los gestores y al personal técnico como intercambiables en tanto sus aptitudes lo permitan; en particular, uno quiere ser capaz de mover al personal fácilmente entre los roles técnicos y de gestión.

11.17 Las barreras para una organización eficaz de doble jerarquía son sociológicas, y deben ser enfrentadas con energía y vigilancia permanente.

11.18 Es fácil fijar escalas salariales similares para los peldaños correspondi-

entes en la escalera de doble jerarquía, pero esto requiere fuertes medidas proactivas para darles el prestigio correspondiente: oficinas similares, servicios de apoyo similares, acciones de gestión sobrecompensadas.

11.19 La organización como un equipo quirúrgico es un ataque radical en todos los aspectos de este problema. Realmente es la respuesta de largo plazo al problema de la organización flexible.

Dos Pasos Adelante y Uno Atrás – Mantenimiento del Programa

11.20 El mantenimiento del programa es esencialmente diferente al mantenimiento del hardware; principalmente consta de cambios que corrigen defectos de diseño, de adición de funcionalidades incrementales, o de adaptaciones a cambios en el ambiente de uso o la configuración.

11.21 El costo total de mantenimiento durante la vida útil de un programa ampliamente usado es normalmente del 40 por ciento o más del costo de su desarrollo.

11.22 Al costo de mantenimiento le afecta considerablemente el número de usuarios. Más usuarios encuentran más errores.

11.23 Campbell muestra una curva interesante de caída y ascenso en los errores por mes a lo largo de la vida útil del producto.

11.24 Corregir un defecto tiene una probabilidad considerable (20 a 50 por ciento) de introducir otro.

11.25 Después de cada reparación, se debe ejecutar el banco entero de casos de pruebas previamente ejecutado en contra de un sistema para asegurar que no ha sido dañado de alguna forma desconocida.

11.26 Los métodos de diseño de programas para eliminar o al menos destacar los efectos laterales pueden lograr una inmensa recompensa en los costos de mantenimiento.

11.27 También se puede lograr con los métodos de implementación de diseños con menos personas, menos interfaces y menos errores.

Un Paso Adelante y Otro Atrás – La Entropía del Sistema Aumenta a lo Largo del Tiempo de Vida Útil

11.28 Lehman y Belady hallan que la cantidad total de módulos se incrementa

linealmente con el número de liberación de un sistema operativo grande (OS/360), pero que la cantidad de módulos afectados se incrementa exponencialmente con el número de liberación.

11.29 Todas la reparaciones tienden a destruir la estructura, a incrementar la entropía y el desorden de un sistema. Incluso el mantenimiento más hábil del programa sólo retrasa su hundimiento en un caos irreparable, a partir del cual tendrá que haber un rediseño a partir de cero. [Muchas de las necesidades reales por actualizar un programa, tales como el rendimiento, atacan principalmente los límites de su estructura interna. A menudo esos límites originales ocasionaron las deficiencias que surgieron más tarde.]

Capítulo 12. Herramientas Afiladas

12.1 El gestor de un proyecto debe establecer una filosofía y asignar recursos para la construcción de herramientas comunes, y al mismo tiempo reconocer la necesidad por herramientas especializadas.

12.2 Los equipos constructores de sistemas operativos necesitan una máquina objetivo propia donde depurar; debe tener la máxima memoria en lugar de la máxima velocidad, y un progamador de sistemas que mantenga el software estándar actualizado y en buen estado.

12.3 También se debe adaptar la máquina de depurado, o su software, tal que se pueda realizar automáticamente cuentas y mediciones de todo tipo de parámetros del programa.

12.4 La demanda por el uso de la máquina destino tiene una peculiar curva de crecimiento: a una baja actividad le sigue un crecimiento explosivo, y luego se estabiliza.

12.5 Depurar el sistema es como la astronomía, siempre se ha hecho principalmente en la noche.

12.6 Asignar bloques considerables de tiempo de la máquina destino a un subequipo a la vez probó ser la mejor manera de organizar, mucho mejor que intercalar el uso entre los subequipos, a pesar de la teoría.

12.7 Este método, el favorito para organizar las escasas computadoras por medio de bloques, ha sobrevivido a 20 años [en 1975] de cambio tecnológico porque es más productivo. [Todavía lo es, en 1995].

12.8 Si la computadora destino es nueva, se necesita un simulador lógico

para ella. Obtenerlo *cuanto antes*, le proporcionará un vehículo de depurado *confiable* incluso después de tener una máquina real.

12.9 Una biblioteca del programa principal debe dividirse en (1) un conjunto de corralitos individuales, (2) una sub biblioteca de integración del sistema, en el presente bajo prueba del sistema y (3) una versión liberada. La separación formal y la evolución dan el control.

12.10 La herramienta que ahorra la mayor cantidad de trabajo en un proyecto de programación es probablemente un sistema de edición de texto.

12.11 La voluminosidad de la documentación del sistema introduce en efecto un nuevo tipo de incomprensibilidad [véase Unix, por ejemplo], aunque es por mucho preferible a la tan común severa falta de documentacion.

12.12 Construya un simulador de rendimiento, desde el punto de vista del usuario y de forma descendente. Inícielo muy pronto. Escúchele cuando hable.

Lenguaje de Alto Nivel

12.13 Sólo la pereza y la inercia impiden la adopción universal del lenguaje de alto nivel y de la programación interactiva. [Actualmente han sido adoptados universalmente]

12.14 El lenguaje de alto nivel mejora no sólo la productividad sino también la depuración; menos errores y más fáciles de encontrar.

12.15 Las clásicas objeciones acerca de la funcionalidad, del espacio del código objeto y la velocidad del código objeto se han hecho obsoletas debido al avance en la tecnología de lenguajes y compiladores.

12.16 El único candidato razonable para la programación de sistemas hoy es PL/I. [Ya no es cierto.]

Programación Interactiva

12.17 Los sistemas interactivos nunca desplazarán a los sistemas por lotes en ciertas aplicaciones. [Todavía es cierto.]

12.18 La depuración es la parte difícil y lenta de la programación de sistemas, y el lento tiempo de respuesta es la ruina de la depuración.

12.19 Una evidencia limitada muestra que la programación interactiva al menos duplica la productividad en la programación de sistemas.

El Todo y las Partes

13.1 El trabajo arquitectónico detallado, y meticulos implícitos en los Capítulos 4, 5 y 6 no sólo facilita el uso, también facilita la construcción y reduce el número de errores del sistema.

13.2 Vyssotsky afirma: "Demasiadas fallas se refieren exactamente a esos aspectos que nunca fueron totalmente especificados."

13.3 Mucho antes de cualquier código en sí, se debe entregar la especificación a un grupo de prueba externo para el escrutinio de su completitud y claridad. Los propios desarrolladores no pueden hacerlo. (Vyssotsky)

13.4 "El diseño descendente de Wirth [un refinamiento a pasos] es la nueva formalización de la programación más importante de la década [1965-1975]."

13.5 Wirth aboga por el uso de una notación de tan alto nivel como sea posible en cada paso.

13.6 Un buen diseño descendente evita errores de cuatro formas.

13.7 Algunas veces se tiene que retroceder, descartar un nivel alto, y empezar de nuevo.

13.8 La programación estructurada, el diseño de programas, cuyas estructuras de control constan sólo de un conjunto específico que gobierna bloques de código (en contra de una variedad de bifurcaciones), es una manera sólida de evitar errores y es la manera correcta de pensar.

13.9 Los resultados experimentales de Gold muestran un progreso tres veces mayor en la primera interacción de una sesión de depurado interactivo como en las interacciones subsecuentes. Todavía vale la pena planificar cuidadosamente la depuración antes de iniciar una sesión. [Creo que *aún* lo hace, en 1995.]

13.10 Me parece que el uso apropiado de un buen sistema [depurado interactivo de respuesta rápida] requiere dos horas de escritorio por cada dos horas de sesión de máquina: una hora para limpiar y documentar después de cada sesión y otra para planificar cambios y pruebas para la próxima sesión.

13.11 La depuración del sistema (en contraste a la depuración de componentes) tomará más tiempo del esperado.

13.12 La dificultad de la depuración del sistema justifica un enfoque comple-

tamente sistemático y planificado.

13.13 Se debería empezar a depurar el sistema sólo depués de que las piezas parecen funcionar (en contra de atornillarlo todo y probar para que emerjan los errores de la interfaz; y en contra de empezar a depurar el sistema cuando se conocen totalmente los errores de los componentes pero no están corregidos.) [Esto es particularmente cierto para los equipos.]

13.14 Vale la pena construir muchos andamios de depuración y códigos de prueba, quizá incluso un 50 por ciento más que el producto que se está depurando.

13.15 Uno debe controlar cambios y versiones, con miembros del equipo que trabajan en copias del corralito.

13.16 Añadir un componente a la vez durante la depuración del sistema.

13.17 Lehman y Belady ofrecen pruebas de que los cuantos de cambio deberían ser grandes e infrecuentes o muy pequeños y frecuentes. Este último está más sujeto a inestabilidad. [Un equipo de Microsoft hace que funcionen cuantos pequeños y frecuentes. El sistema en desarrollo se reconstruye cada noche.]

Capítulo 14. Incubando una Catástrofe

14.1 "¿Cómo es que un proyecto se retrasa un año? . . . Un día a la vez."

14.2 El retraso diario es más difícil de reconocer, más difícil de evitar y más difícil de recuperar que las calamidades.

14.3 El primer paso para controlar un proyecto grande en un calendario ajustado es *tener* un calendario, construido de hitos y fechas para ellos .

14.4 Los hitos deben ser concretos, específicos, sucesos medibles, definidos con el filo de una navaja.

14.5 Si el hito es muy estricto tal que nadie pueda engañarse, rara vez el programador mentirá acerca del avance del hito.

14.6 Los estudios del comportamiento de la estimación de los contratistas del gobierno en proyectos grandes muestran que la estimación del tiempo de actividad revisada cuidadosamente cada dos semanas no cambia de forma signifiva en tanto se aproxima el momento de incio, que durante la actividad la sobreestimación de la actividad decrece constantemente; y que la *subestimación* no cambia hasta alrededor de tres se-

manas antes de la conclusión programada.

14.7 Un retraso crónico del calendario es un asesino moral. [Jim Mc-Carthy de Microsoft dice, "Si usted no cumple con una fecha, asegúrese de cumplir la siguiente."[2]]

14.8 El *empuje* es esencial para los equipos de programación, así como para los grandes equipos de béisbol.

14.9 No existe sustituto para un calendario de camino crítico que nos permita decir qué retraso importa y cuánto.

14.10 La preparación de un diagrama de camino crítico es la parte más valiosa de su uso, puesto que diseñar la red, identificar las dependencias, y estimar los segmentos obliga muy temprano a una gran cantidad de planificación muy específica dentro de un proyecto.

14.11 El primer diagrama es siempre terrible, y uno inventa y reinventa preparando el próximo.

14.12 Un diagrama de camino crítico responde a la excusa desmoralizante, "De cualquier manera, la otra pieza está retrasada."

14.13 Todo jefe necesita tanto información de excepciones que requieren acción y un panorama del estado como formación y una temprana advertencia futura.

14.14 Obtener el estado es difícil, puesto que los gestores subordinados tienen toda la razón de no compartirlo.

14.15 A través de una mala acción, un jefe puede garantizar acallar completamente la revelación del estado, al contrario, separando cuidadosamente los informes del estado y aceptándolos sin pánico o apropiación fomentará informes honestos.

14.16 Uno debe tener técnicas de revisión mediante los cuales el personal conozca el estado verdadero. Para este propósito la clave es un calendario de hitos y un documento de finalización.

14.17 Vyssotsky: "Me ha parecido práctico llevar tanto las fechas del 'calendario' (fechas del jefe) como las estimadas (fechas del gestor de menor nivel) en el informe de hitos. El gestor de proyecto tiene que mantener sus manos fuera de las fechas estimadas."

14.18 Un pequeño equipo de *Planes y Controles* que mentenga el informe de hitos es invaluable para un proyecto grande.

Capítulo 15. La otra Cara

15.1 Para el producto del programa, la otra cara hacia el usuario, la documentación, es absolutamente tan importante como la cara hacia la máquina.

15.2 Incluso para los programas más privados, la documentación en prosa es necesaria, porque al usuario-autor le fallará la memoria.

15.3 Los maestros y gestores con todo hemos errado en inculcar en los programadores una actitud acerca de la documentación que los inspire por el resto de su vida, sobreponiéndolos a la pereza y a las presiones del calendario.

15.4 El fracaso no se debe tanto a la carencia de celo o elocuencia como al fracaso en mostrar *cómo* documentar eficaz y económicamente.

15.5 La mayor parte de la documentación fracasa en dar una pequeña *visión global*. Retroceda y haga un acercamiento lentamente.

15.6 La documentación crítica del usuario debe ser redactada antes de que se construya el programa, porque plasma las decisiones básicas de planificación. Debería describir nueve cosas (ver el capítulo).

15.7 Un programa debería ser entregado con unos cuantos casos de prueba, unos para datos de entrada válidos, otros para datos de entrada en el margen de la validez, y otras claramente para datos de entrada inválidos.

15.8 La documentación de las cuestiones internas del programa, para la persona que lo debe modificar, también exige una visión general en prosa, la cual debería contener cinco tipos de cosas (ver el capítulo)

15.9 El diagrama de flujo es una de las piezas más completamente sobrevaloradas de la documentación del programa; el diagrama de flujo minucioso y pormenorizado es una molestia, vuelto obsoleto por la *escritura* en lenguajes de alto nivel. (Un diagrama de flujo es un lenguaje de alto nivel *diagramado*.)

15.10 Pocos programas necesitan un diagrama de flujo de más de una página, si acaso. [Los requisitos de la documentación MILSPEC están realmente equivocados en este punto.]

15.11 Asimismo se necesita un grafo de la estructura del programa, que no requiere los estándares del diagramado de flujo de la ANSI.

15.12 Para preservar el mantenimiento de la documentación, es crucial que

sea incorporada en el programa fuente, en lugar de preservarla como un documento separado.

15.13 Tres ideas son clave para minimizar la carga de la documentación:

- Usar partes del programa que tienen que estar de cualquier forma, tales como nombres y declaraciones, portar tanta documentación como sea posible.
- Usar el espacio y formato para mostrar la subordinación y el anidamiento y mejorar la legibilidad.
- Insertar la documentación necesaria en prosa dentro del programa como párrafos del comentarios, especialmente como encabezados de módulos.

15.14 En la documentación para el uso de los modificadores del programa, diga *por qué* las cosas son como son, en lugar de únicamente cómo son. El *propósito* es la clave para entender; incluso la sintaxis del lenguaje de alto nivel no transmite propósito en absoluto.

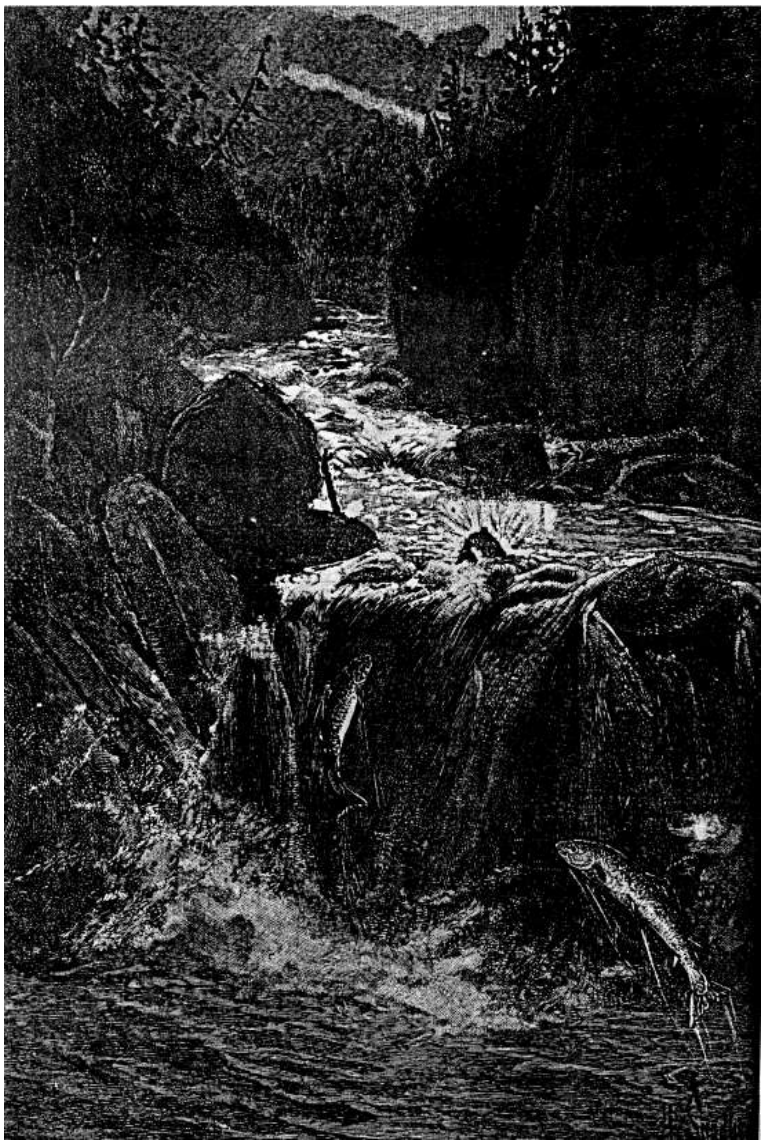
15.15 Las técnicas de programación autodocumentadas hallan su mejor uso y poder en los lenguajes de alto nivel utilizados con sistemas en línea, que son las herramientas que se *deberían* usar.

Epílogo Original

E.1 Los sistemas de software son tal vez las cosas más intrincadas y complejas (en términos del número de distintos tipos de partes) que la humanidad construye.

E.2 El pozo de brea de la ingeniería del software continuará siendo pegajoso por mucho tiempo.

19
El Mítico Hombre-Mes: 20 Años después



19
El Mítico Hombre-Mes: 20 Años después

No conozco otra manera de juzgar el futuro que no sea por el pasado.

PATRICK HENRY

No se puede planear el futuro a través del pasado.

EDMUND BURKE

Por Qué una Edición por el Vigésimo Aniversario?

El avión zumbaba a través de la noche hacia LaGuardia. Las nubes y la obscuridad velaban todas las vistas interesantes. El documento que estaba examinando era tedioso. Sin embargo, yo no estaba aburrido. El extraño sentado a mi lado estaba leyendo *El Mítico Hombre-Mes*, y yo estaba esperando a ver si por una palabra o gesto reaccionaría. Finalmente cuando empezamos a rodar por la pista hacia la entrada, no pude esperar más:

> "¿Qué tal el libro? ¿Lo recomienda?"
>
> "Mmm! No hay nada en él que no lo supiera ya."

Decidí no presentarme.

¿Por qué ha persistido *El Mítico hombre-Mes?* ¿Por qué hoy todavía parece ser relevante en la práctica del software? ¿Por qué tiene lectores fuera de la comunidad de la ingeniería de software, generando estudios, citas, y cartas de abogados, doctores, psicólogos, sociólogos, como también de gente del software? ¿Cómo puede un libro escrito hace 20 años sobre la experiencia de la construcción del software de hace 30 años todavía ser relevante y mucho menos útil?

Una explicación que a veces escucho es que la disciplina del desarrollo del software no ha avanzado de forma normal o apropiada. Esta visión generalmente está sustentada al contrastar la productividad del desarrollo del software de computadoras con la productividad de la manufactura del hardware, que se ha multiplicado por lo menos por mil a lo largo de dos décadas. Como se explica en el Capítulo 16, la anomalía no es que el software haya tenido un progreso tan lento, sino más bien que la tecnología de las computadoras haya explotado de una manera sin par en la historia de la humanidad. En líneas generales esto viene de la transición gradual de la manufactura de computadoras a partir de una industria del ensamblaje a una industria de procesos, de una manufactura intensiva en mano de obra a una manufactura intensiva en capital. En contraste a la manufactura, el desarrollo del hardware y el software permanecen intrínsecamente intensivos en mano de obra.

Una segunda explicación promovida con frecuencia es que *El Mítico Hombre-Mes* sólo trata incidentalmente acerca del software más bien trata principalmente acerca de cómo construimos cosas en equipos. Seguramente hay

algo de verdad en esto; el prefacio de la edición de 1975 afirma que la gestión de un proyecto de software es más parecido a otras gestiones de lo que inicialmente creían los programadores. Todavía pienso que eso es cierto. La historia de la humanidad es un teatro en el cual los relatos siguen siendo los mismos, los libretos de esos relatos cambian lentamente conforme las culturas evolucionan y los escenarios cambian todo el tiempo. Así es como vemos nuestro propio siglo veinte reflejado en Shakespeare, Homero y la Biblia. Así, en la medida en que *El MH-M* trata acerca de la gente y los equipos, la obsolescencia debería ser lenta.

Cualquiera que sea el motivo, los lectores siguen comprando el libro, y continúan enviándome muchos comentarios valiosos. Hoy en día a menudo me preguntan, "¿Qué piensa ahora que estuvo mal cuando lo escribió? ¿Qué se ha vuelto hoy en día obsoleto? ¿Qué es lo realmente nuevo en el mundo de la ingeniería de software?" Estas preguntas bastante diferentes son todas razonables, y las abordaré lo mejor que pueda. Sin embargo, no en ese orden, sino en grupos de tópicos. Primero, vamos a considerar lo que fue correcto cuando se escribió, y que aún lo sea.

El Argumento Central: La Integridad Conceptual y el Arquitecto

La integridad conceptual. Un producto de programación limpio y elegante debe mostrar a cada uno de sus usuarios un modelo mental coherente de la aplicación, de las estrategias para realizar la aplicación y de las tácticas de la interfaz de usuario utilizadas en acciones específicas y parámetros. La integridad conceptual del producto, como la percibe el usuario, es el factor más importante de la facilidad de uso. (Por supuesto, existen otros factores. La uniformidad de la interfaz de usuario de la Macintosh a lo largo de todas las aplicaciones es un ejemplo importante. Es más, es posible construir interfaces coherentes que no obstante sean bastante incómodas. Considere MS-DOS.)

Existen muchos ejemplos de productos de software elegantes diseñados por una sola mente, o por un par. La mayoría de los trabajos puramente intelectuales tales como libros o composiciones musicales se producen así. Sin embargo, los procesos de desarrollo de productos en muchas industrias no pueden permitirse este enfoque directo de la integridad conceptual. Las presiones competitivas obligan a actuar con urgencia; en muchas tecnologías

modernas el producto final es bastante complejo, y el diseño intrínsecamente requiere muchos hombre-meses de esfuerzo. Los productos de software son a la vez complejos y ferozmente competitivos respecto al calendario.

Aquí tropezamos con una dificultad peculiar, porque para cualquier producto que sea suficientemente grande o urgente se requiere el esfuerzo de muchas mentes: el resultado debe ser conceptualmente coherente para la mente de un usuario y, al mismo tiempo, debe ser diseñado por muchas mentes. Cómo se organiza la labor de diseño para conseguir dicha integridad conceptual? Esta es la principal pregunta abordada por *El MH-M*. Una de sus tesis es que la gestión de proyectos de programación grandes es cualitativamente diferente a la gestión de pequeños proyectos, debido sólo al número de mentes involucradas. Para conseguir la coherencia se necesitan acciones de gestión cuidadosas, e incluso heroicas.

El arquitecto. En los Capítulos del 4 al 7 afirmo que la acción más importante es el encargo a una sola mente como el *arquitecto* del producto, que es responsable por la integridad conceptual de todos los aspectos perceptibles del producto por el usuario. El arquitecto da forma y posee el modelo mental público del producto que será utilizado para explicarle al usuario cómo usarlo. Esto incluye la especificación detallada de toda su funcionalidad y la manera de invocarla y controlarla. El arquitecto es también el agente del usuario, representa eruditamente el interés del usuario en los inevitables compromisos entre la funcionalidad, el rendimiento, el tamaño, el costo y el calendario. Este papel es un trabajo de tiempo completo, y solo en los equipos más pequeños puede combinarse con el de gestor de equipo. El arquitecto es como el director y el gestor como el productor de la película.

Separar la arquitectura de la implementación y la realización. Para lograr que la tarea crucial del arquitecto sea siquiera concebible, es necesario separar la arquitectura, la definición del producto como el usuario lo percibe, de su implementación. Entre arquitectura e implementación se define una frontera limpia entre los distintas segmentos de la tarea de diseño, y hay trabajo de sobra en ambos lados.

Recursión de arquitectos. Para productos bastante grandes, una mente no

puede elaborar toda la arquitectura, incluso después de que todo lo concerniente a la implementación se haya dividido. Así que es necesario para el arquitecto principal dividir el sistema en subsistemas. Las fronteras del subsistema deben estar en aquellos lugares donde las interfaces entre los subsistemas sean mínimas y más fáciles de definir rigurosamente. Después cada pieza tendrá su propio arquitecto, que debe informar al arquitecto principal del sistema en relación a la arquitectura. Claramente este proceso puede avanzar recursivamente según sea necesario.

Hoy en día estoy más convencido que nunca. La integridad conceptual es central a la calidad del producto. Tener un arquitecto del sistema *es* el paso más importante hacia la integridad conceptual. Estos principios no se limitan de ninguna manera a los sistemas de software, sino al diseño de cualquier construcción compleja, ya sea una computadora, un avión, una Iniciativa de Defensa Estratégica o un Sistema de Posicionamiento Global. Después de enseñar el curso de laboratorio de ingeniería de software más de veinte veces, llegué a insistir en que los equipos de estudiantes tan pequeños como de cuatro personas escogieran un gestor y aparte un arquitecto. Definir distintos papeles en equipos tan pequeños puede ser un poco radical, pero he observado que funciona bien y contribuye a un diseño exitoso incluso en equipos pequeños.

El efecto del Segundo Sistema. Caracteristis y Suposiciones de Frecuencia

Diseño para grandes grupos de usuarios. Una de las consecuencias de la revolución de las computadoras personales es que cada vez más, al menos en la comunidad del procesamiento de datos de negocios, los paquetes comerciales están reemplazando a las aplicaciones personalizadas. Además, los paquetes de software estándar venden cientos de miles de copias, e incluso millones. Los arquitectos de sistemas para el software suministrado por los vendedores de computadoras siempre tuvieron que diseñar para un conjunto de usuarios grande y amorfo en lugar de para una aplicación única y definible en una compañía. Ahora existen bastantes arquitectos de sistemas que encaran esta tarea.

Paradójicamente, es mucho más difícil diseñar una herramienta de propó-

sito general que diseñar una herramienta de propósito especial, precisamente porque se tiene que asignar pesos a las diferentes necesidades de los diversos usuarios.

Caracteritis. La tentación que acosa al arquitecto de una herramienta de propósito general tal como una hoja de cálculo o un procesador de palabras es sobrecargar el producto con características de utilidad marginal, a expensas del rendimiento e incluso de la facilidad de uso. El atractivo de las características propuestas es evidente desde el principio; el castigo en el rendimiento solo es evidente a medida que avanzan las pruebas del sistema. La pérdida de la facilidad de uso se acerca con sigilo maliciosamente, a medida que se van añadiendo características en pequeñas dosis, y los manuales se ponen más y más gordos.[1]

Para los productos del mercado masivo que sobreviven y evolucionan a través de varias generaciones, la tentación es especialmente fuerte. Millones de consumidores solicitan cientos de características; cualquier solicitud es por sí misma prueba de que "el mercado lo exige". Con frecuencia, el arquitecto del sistema original se ha marchado por mayores glorias, y la arquitectura está en manos de gente con menos experiencia en representar el interés global del usuario con equidad. Un reciente estudio de Microsoft Word 6.0 dice "El Word 6.0 atestado de características; la actualización ralentizada por el equipaje . . . El Word 6.0 es también grande y lento." Se nota con disgusto que el Word 6.0 requiere 4 MB de RAM, y continúa diciendo que la rica funcionalidad agregada significa que "incluso una Macintosh IIfx [está] apenas a la altura del desempeño de Word 6."[2]

Definiendo el grupo de usuarios. Cuanto más grande y amorfo es el grupo de usuarios, se hace más necesario definirlo explícitamente para lograr la integridad conceptual. Cada miembro del equipo de diseño seguramente tendrá una imagen mental implícita de los usuarios, y la imagen de cada diseñador será diferente. Puesto que la imagen que tiene el arquitecto del usuario consciente o subconscientemente afecta cada decisión arquitectónica, es importante que un equipo de diseño arribe a una sola imagen compartida por todos. Y eso requiere anotar los atributos del grupo de usuarios supuestos, incluyendo:

- Quiénes son
- Qué necesitan
- Qué creen que necesitan
- Qué quieren

Frecuencias. Para cualquier producto de software, cualquiera de los atributos del conjunto de usuarios es de hecho una distribución, con muchos valores posibles, cada uno con su propia frecuencia. ¿Cómo es que el arquitecto obtiene estas frecuencias? Inspeccionar esta población mal definida es una propuesta dudosa y costosa.[3] A lo largo de los años me he convencido que un arquitecto debe *adivinar*, o, si se prefiere, *postular*, un conjunto completo de atributos y valores con sus frecuencias, con el fin de desarrollar una descripción completa, explícita y compartida del conjunto de usuarios.

Muchos beneficios derivan de este procedimiento insólito. Primero, el cuidadoso proceso de adivinación de las frecuencias hará que el arquitecto piense muy cuidadosamente acerca del supuesto conjunto de usuarios. Segundo, al anotar las frecuencias las someterá a debate, lo cual aclarará a todos los participantes y emergerán las diferencias en las imágenes del usuario que los distintos diseñadores arrastran. Tercero, enumerar las frecuencias explícitamente ayuda a todos a reconocer qué decisiones dependen de qué propiedades del conjunto de usuarios. Incluso este tipo de análisis de sensibilidad informal es valioso. Cuando revela que decisiones muy importantes dependen de alguna conjetura en particular, entonces vale la pena el costo de establecer mejores estimaciones para ese valor. (El sistema gIBIS desarrollado por Jeff Conklin provee una herramienta para realizar un seguimiento formal y exacto de las decisiones de diseño y la documentación de los motivos de cada una.[4] No he tenido la oportunidad de usarlo, pero creo que sería muy útil.)

Resumiendo: anotar explícitamente las conjeturas de los atributos del conjunto de usuarios. *Es mucho mejor ser una persona explícita aunque equivocada que ser vago.*

¿Qué hay acerca del "Efecto del Segundo Sistema"? Un estudiante perspicaz comentó que *EL Mítico Hombre-Mes* recomendaba una receta para el desastre: Planifique liberar la segunda versión de cualquier sistema (Capítulo 11), lo cual en el Capítulo 5 se caracteriza como el sistema más peligroso que

alguien jamás diseña. Tuve que concederle un "te descubrí."

La contradicción es más lingüística que real. El "segundo" sistema descrito en el Capítulo 5 es el segundo sistema en uso, el sistema resultante que induce a añadir funcionalidades y adornos. El "segundo" sistema del Capítulo 11 es el segundo intento de construir lo que debería ser el primer sistema puesto en uso. Está construido bajo todas las restricciones del calendario, talento e ignorancia que caracterizan a los nuevos proyectos – las restricciones que ejercen una escasa disciplina.

El Triunfo de la Interfaz VIMP

Uno de los desarrollos más impresionantes del software durante las dos décadas pasadas ha sido el triunfo de la interfaz Ventana, Iconos, Menús, Puntero de interfaz – o VIMP para abreviar. Hoy en día es tan familiar que no necesita descripción. Este concepto fue expuesto públicamente por primera vez por Doug Englebart y su equipo del Stanford Research Institute en el congreso Western Joint Computer Conference de 1968.[5] De allí, estas ideas se trasladaron al Xerox Palo Alto Research Center, de donde emergió en la estación de trabajo personal Alto desarrollada por Bob Taylor y su equipo. Luego fueron contratados por Steve Jobs para la Apple Lisa, una computadora demasiado lenta para soportar sus emocionantes conceptos de facilidad de uso. Después Jobs plasmó estos conceptos en la comercialmente exitosa Apple Macintosh en 1985. Y fueron adoptados más tarde por Microsoft Windows en la IBM PC y compatibles. La versión de Mac será mi ejemplo. [6]

La integridad conceptual a través de una metáfora. La VIMP es un espléndido ejemplo de una interfaz de usuario que tiene integridad conceptual, conseguida a través de la adopción de un modelo mental habitual, la metáfora del escritorio, y su cuidadosa y consistente extensión para explotar una implementación de gráficas por computadora. Por ejemplo, la apropiada aunque costosa decisión de superponer ventanas en lugar de un tapizado de ellas se deriva directamente de la metáfora. La capacidad de dimensionar y dar forma a las ventanas es una extensión consistente que brinda al usuario los nuevos poderes habilitados a través de gráficas por computadora. Los papeles sobre el escritorio no pueden ser tan fácilmente dimensionados ni moldeados. El arrastrar y soltar proviene directamente de la metáfora; se-

leccionar íconos apuntando con un cursor es una analogía directa de recoger cosas con la mano. Los íconos y las carpetas anidadas son analogías fieles de los documentos de escritorio; al igual que el bote de basura. Los conceptos de cortar, copiar y pegar reflejan fielmente las cosas que solemos hacer con los documentos en los escritorios. Seguimos tan fielmente la metáfora y es tan consistente su extensión que los nuevos usuarios se sorprenden positivamente por la noción de arrastre de un icono de disquete a la basura para expulsar el disco. Si la interfaz no fuera consistente de forma casi uniforme, esa inconsistencia (bastante mala) no rechinaría mucho.

¿Dónde se obliga a la interfaz VIMP a ir más allá de la metáfora de la interfaz? Principalmente en dos aspectos: los menús y una mano dominante. Cuando trabajamos con un escritorio real, se *realizan* acciones en los documentos, en vez de decirle a alguien o a algo que los haga. Y cuando uno le dice a alguien que realice una acción, por lo general uno produce, en lugar de seleccionar, los comandos verbales imperativos orales o escritos: "Por favor archive esto." "Por favor encuentre la correspondencia anterior." "Por favor envíe esto para que Mary lo maneje."

Desgraciadamente, la interpretación confiable de los comandos en Inglés generados de forma libre está más allá del actual estado del arte, ya sea que los comandos sean escritos o hablados. Por lo tanto, los diseñadores de la interfaz fueron eliminados en dos pasos a partir de la acción directa del usuario en los documentos. Sabiamente recogieron del escritorio habitual su único ejemplo de selección de comandos – la hoja de ruta impresa, en la que el usuario selecciona de entre un menú restringido de comandos cuya semántica está estandarizada. Esta idea la extendieron a un menú horizontal de submenús verticales desplegables.

Declaraciones de comandos y el problema de los dos cursores. Los comandos son oraciones imperativas; siempre tienen un verbo y generalmente un objeto directo. Para cualquier acción, se necesita especificar un verbo y un sustantivo. La metáfora del puntero dice, especificar dos cosas al mismo tiempo, tener dos cursores diferenciados en la pantalla, cada uno manejado por un ratón distinto – uno en la mano derecha y otro en la izquierda. Después de todo, en un escritorio físico normalmente trabajamos con ambas manos. (Pero, una mano suele mantener cosas fijas en su lugar, lo cual ocurre por

omisión en el escritorio de la computadora.) La mente es desde luego capaz de realizar operaciones con las dos manos; con regularidad usamos dos manos para teclear, manejar, cocinar. Desgraciadamente, el suministro de un ratón ya era un gran avance para los fabricantes de computadoras personales; ningún sistema comercial admite dos acciones simultáneas del cursor del ratón, cada uno manejado con una mano distinta.[7]

Los diseñadores de la interfaz aceptaron la realidad y la diseñaron para un solo ratón, adoptando la convención sintáctica de que uno primero apunta (*selecciona*) el sustantivo. Luego se apunta al verbo, que es un elemento del menú. Esto realmente traiciona mucho la facilidad de uso. Mientras veo a los usuarios, o las cintas de video de los usuarios, o el trazado en pantalla de los movimientos del cursor, me sorprendo inmediatamente que un solo cursor tenga que hacer el trabajo de dos: seleccionar un objeto en la parte del escritorio de la ventana; escoger un verbo en la parte del menú; buscar o volver a encontrar un objeto en el escritorio; desplegar nuevamente un menú (generalmente el mismo) y escoger un verbo. El cursor se mueve de acá para allá, de allá para acá, del espacio de datos al espacio de menú, desechando cada vez información útil acerca de dónde estaba la última vez que estuvo en este espacio – en general, es un proceso ineficiente.

Una solución brillante. Incluso si la electrónica y el software pudieran manejar fácilmente dos cursores activos simultáneamente, existen dificultades de distribución de espacio. El escritorio en la metáfora VIMP realmente incluye una máquina de escribir, y uno debe acomodar su teclado real en el espacio físico en un escritorio real. Un teclado más dos tapetes de ratón usan gran parte de los bienes a nuestro alcance. Ahora bién, el problema del teclado puede convertirse en una oportunidad – por qué no habilitar la operación eficiente de dos manos usando una mano en el teclado para especificar verbos y la otra en un ratón para seleccionar sustantivos. Ahora el cursor yace en el espacio de datos, explotando la elevada localidad de sucesivas selecciones de sustantivos. Una eficiencia real implica un auténtico poder del usuario.

Poder del usuario versus facilidad de uso. Esa solución, sin embargo, traiciona lo que hace que los menús sean muy fáciles de usar por los novatos – los menús presentan los verbos alternativos válidos en cualquier estado en

particular. Podemos comprar un paquete, traerlo a casa, y empezar a usarlo sin consultar el manual, simplemente sabiendo que lo compramos por, y para experimentar con los diferentes verbos del menú.

Una de las cuestiones más difíciles que encaran los arquitectos de software es precisamente cómo equilibrar el poder del usuario frente a la facilidad de uso. ¿Están diseñados para una operación sencilla del novato o usuario ocasional, o para la poderosa eficiencia del usuario profesional? La respuesta ideal es proporcionar ambas, de una forma conceptualmente coherente – esto se consigue en la interfaz VIMP. Los verbos del menú de alta frecuencia tienen cada uno equivalentes de tecla-única + clave-del-comando, en su mayoría seleccionados tal que puedan ser fácilmente pulsados como una solo acorde con la mano izquierda. Por ejemplo, en la Mac la tecla de comando (⌘) está justo debajo de las teclas Z y X, por lo tanto, las operaciones de alta frecuencia se codifican como ⌘z, ⌘x, ⌘c, ⌘v, ⌘s.

Transición incremental de novato a usuario experto. Este sistema dual para especificar verbos de comandos no sólo satisface la necesidad de un bajo esfuerzo de aprendizaje del novato y la necesidades de eficiencia del usuario avanzado; sino que también permite a cada usuario una transición suave entre modos. Las codificaciones de letras, llamadas *atajos*, aparecen en los menús junto a los verbos, tal que un usuario en duda puede activar el menú para verificar la letra equivalente, en lugar de simplemente seleccionar el elemento del menú. Todo novato aprende primero los atajos para sus propias operaciones de alta frecuencia. Puede probar cualquier atajo acerca del cual tenga duda, puesto que ⌘z deshará cualquier error. O bien, puede revisar el menú para ver qué comandos son válidos. Los novatos desplegarán muchos menús; los usuarios avanzados muy pocos; y los usuarios intermedios que solo ocasionalmente necesitarán seleccionar desde un menú, puesto cada uno conocerá los pocos atajos que componen la mayoría de sus propias operaciones. La mayoría de nosotros, diseñadores de software, estamos muy familiarizados con esta interfaz para apreciar plenamente su elegancia y poder.

El éxito de la incorporación directa como un mecanismo para imponer la arquitectura. La interfaz de la Mac es todavía notable en otra forma. Sin coerción, sus diseñadores la han convertido en una interfaz estándar en todas

las aplicaciones, incluyendo la vasta mayoría que son escritas por terceros. De este modo, el usuario gana coherencia conceptual a nivel de interfaz no sólo dentro del software suministrado con la computadora sino a través de todas las aplicaciones.

Esta hazaña la lograron los diseñadores de la Mac al construir la interfaz en la memoria de solo lectura, de modo que es más fácil y rápido para los desarrolladores usarla que construir sus propias interfaces idiosincrásicas. Estos incentivos naturales por la uniformidad prevalecieron lo suficiente para establecer un estándar *de facto*. Los incentivos naturales fueron ayudados por un compromiso de gestión total y mucha persuasión por parte de Apple. Los revisores independientes de las revistas de productos, reconocieron el inmenso valor de la integridad conceptual de las aplicaciones cruzadas, han complementado también los incentivos naturales a través de la crítica despiadada a productos que no se amoldan.

Este es un magnífico ejemplo de la técnica, recomendada en el Capítulo 6, de conseguir uniformidad fomentando en otros a incorporar directamente nuestro propio código en sus productos, en lugar de intentar hacer que construyan su propio software según nuestras especificaciones.

El destino de la interfaz VIMP: La obsolescencia. A pesar de sus excelencias, espero que la interfaz VIMP sea una reliquia histórica en una generación. El apuntar todavía será la manera de expresar sustantivos en tanto dominemos nuestras máquinas; el habla es seguramente la forma correcta de expresar verbos. Las herramientas como Voice Navigator para Mac y Dragon para PC ya ofrecen esta capacidad.

No Construya para Desechar – El Modelo en Cascada Está Errado

La inolvidable imagen de Galloping Gertie, el Puente de los Estrechos de Tacoma, inicia el Capítulo 11, que recomienda radicalmente: "Planifique desechar; de cualquier manera lo hará." Esto lo percibo ahora como equivocado, no porque sea muy radical, sino porque es muy simplista.

El más grande error en el concepto de "Construir para desechar" es que implícitamente supone el modelo secuencial clásico o en cascada de la construcción de software. El modelo proviene de un diseño en diagrama de Gantt

de un proceso en etapas, y a menudo se dibuja como en la Figura 19.1. Winston Royce mejoró el modelo secuencial en un artículo clásico de 1970 al proporcionar

- Algo de retroalimentación de una etapa hacia sus predecesores.
- Limitar la retroalimentación únicamente a la etapa inmediatamente anterior, para contener el costo y el retraso en el calendario que esto ocasiona.

Precedió a *El MH-M* al aconsejar a los constructores a "construirlo dos veces."[8] El Capítulo 11 no es el único contaminado por el modelo secuencial en cascada; corre a través del libro, empezando con la regla del calendario en el Capítulo 2. Esa regla empírica destina $\frac{1}{3}$ del calendario a la planificación, $\frac{1}{6}$ a la codificación, $\frac{1}{4}$ a la prueba de componentes y $\frac{1}{4}$ a la prueba del sistema.

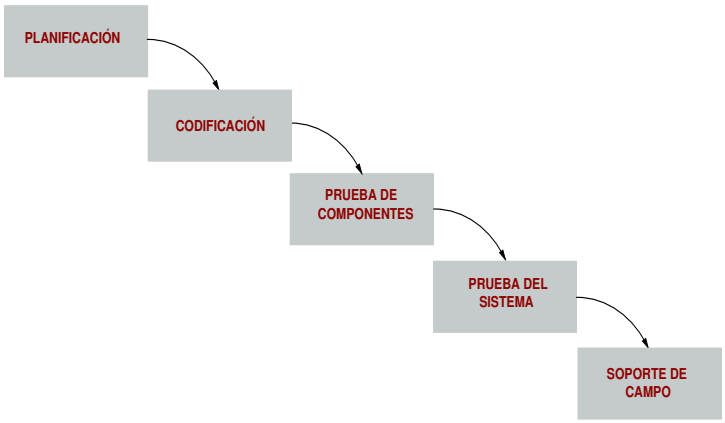


Fig. 19.1 Modelo en cascada de la construcción de software

La principal falacia del modelo en cascada es que supone un proyecto que pasa por el proceso una sola vez, que la arquitectura es excelente y fácil de usar, el diseño de la implementación es sólido, y la realización es corregible a medida que las pruebas avanzan. Otra forma de decirlo es que el modelo en cascada supone que todos los errores estarán en la realización, de tal manera que sus correcciones se pueden entremezclar sin contratiempos con las pruebas de componentes y del sistema.

"Planifique desechar" ataca en efecto frontalmente esta falacia. No es el diagnóstico el que está errado; es el remedio. Ahora he sugerido que se podría descartar y rediseñar el primer sistema pieza por pieza, en vez de hacerlo en un solo bloque. Esto está bien hasta ahora, pero no llega a la raíz del problema. El modelo en cascada coloca la prueba del sistema, y por consiguiente la prueba del *usuario*, al final del proceso de construcción. De este modo, después de dedicarnos a la construcción completa, podemos encontrar incomodidades imposibles para los usuarios, o un rendimiento inaceptable, o una propensión peligrosa al error o malicia del usuario. Por supuesto, el escrutinio de la prueba Alfa de las especificaciones se enfoca a encontrar dichos defectos prematuramente, aunque no existe substituto para los usuarios activos.

La segunda falacia del modelo en cascada es que supone que se construye el sistema completo de una sola vez, se combinan las piezas para una prueba del sistema de punta a punta después de haber hecho todo el diseño de la implementación, la mayor parte del código y gran parte de las pruebas de componentes.

El modelo en cascada, era la forma en que pensaba la mayor parte de la gente acerca de los proyectos de software en 1975, desafortunadamente consagrado en el DOD-STD-2167, la especificación del Departamento de Defensa para el software militar. Esto garantizó su supervivencia aún cuando la mayoría de los practicantes más atentos reconocieron su insuficiencia y lo abandonaron. Afortunadamente, desde entonces, el Departamento de Defensa ha empezado a ver la luz.[9]

Tiene que haber un movimiento aguas arriba. Al igual que el enérgico salmón de la imagen de apertura de este capítulo, la experiencia e ideas de cada parte aguas abajo del proceso de construcción deben hacer saltos aguas

arriba, algunas veces más de una etapa, y afectar la actividad aguas arriba.

Al diseñar la implementación se mostrará que algunos rasgos arquitectónicos perjudican el rendimiento; así que la arquitectura tiene que ser rediseñada. La codificación de la realización mostrará que algunas funcionalidades aumentan los requerimientos de espacio; así que puede que tenga que haber cambios en la arquitectura y la implementación.

Por lo tanto, bien se podría iterar a través de dos o más ciclos de diseño de la arquitectura-implementación antes de llevar a cabo cualquier cosa como la codificación.

Un Modelo de Construcción Incremental Es Mejor – Refinamiento Progresivo

Construir un esqueleto del sistema de punta a punta

Harlan Mills, mientras trabajaba en un entorno de un sistema de tiempo real, abogó, desde un principio, por construir el lazo de sondeo básico de un sistema de tiempo real, con llamadas a subrutinas *(incompletas)* para todas las funciones (Fig 19.2), pero solo subrutinas nulas. Compilarlo y probarlo. Ejecutándose cíclicamente, literalmente sin hacer nada, pero haciéndolo correctamente[10]

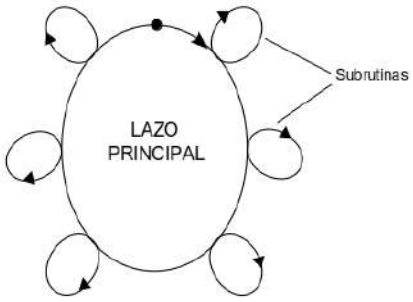


Fig. 19.2

A continuación, desarrollamos un módulo de entrada (quizá uno primitivo) y un módulo de salida. ¡Voilà! Un sistema en ejecución que hace algo, por aburrido que sea. Ahora, función por función, construimos y añadimos módulos de forma incremental. *En cada etapa tenemos un sistema en ejecución.* Si somos diligentes, tenemos en cada etapa un sistema depurado y probado. (A medida que el sistema crece, también lo hace la carga de prueba de regresión de cada nuevo módulo en contra de todos los casos de prueba anteriores.)

Después de que cada función trabaja a un nivel primitivo, refinamos o reescribimos primero un módulo y luego otro y *crecemos* el sistema incrementalmente. Algunas veces, para estar seguros, tenemos que cambiar el lazo de control original, y/o incluso sus interfaces de módulo.

Puesto que tenemos un sistema en funcionamiento todo el tiempo

- podemos comenzar las pruebas de usuario muy pronto, y
- podemos adoptar una estrategia de construir para el presupuesto que brinde protección completamente en contra del calendario o los sobrecostos (a expensas de un posible déficit funcional).

Por espacio de 22 años, enseñé el laboratorio de ingeniería de software en la Universidad de Carolina del Norte, algunas veces conjuntamente con David Parnas. En este curso, generalmente los equipos de cuatro estudiantes construían en un semestre cierto sistema de aplicación de software real. Como a la mitad de camino de esos años, me cambié a enseñar desarrollo incremental. Me quedé sorprendido por el efecto electrizante en la moral de los equipos de esa primera imagen en la pantalla y de ese primer sistema en ejecución.

Familias de Parnas

David Parnas ha sido un importante líder de pensamiento en ingeniería de software durante todo este periodo de 20 años. Todos están familiarizados con su concepto de ocultamiento de información. Algo menos familiar, aunque muy importante, es el concepto de Parnas del diseño de un producto de software como una *familia* de productos relacionados.[11] Exhorta al diseñador a anticipar por un lado las extensiones laterales y por otro las versiones sucesivas de un producto, y definir su funcionalidad o diferencias de plataforma así como construir un árbol genealógico de productos relacionados (Fig. 19.3).

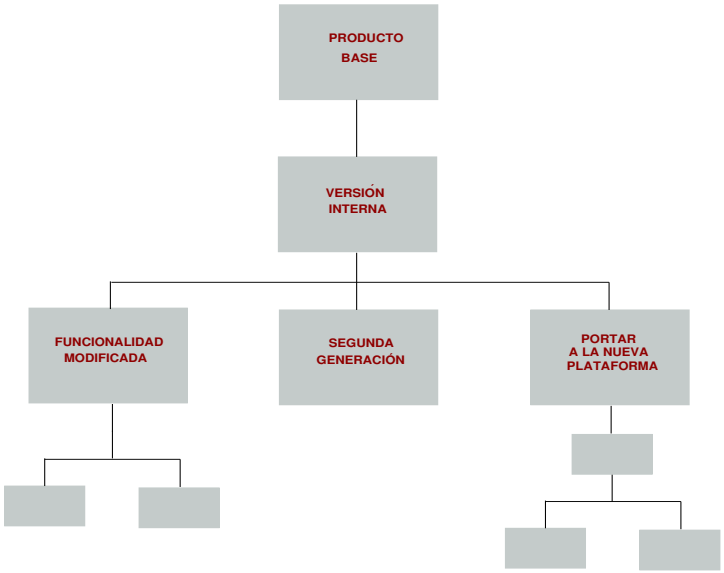


Fig. 19.3

El truco en el diseño de dicho árbol consiste en poner cerca de sus raíces esas decisiones de diseños que tienen menos probabilidades de cambiar.

Esta estrategia de diseño maximiza el reuso de módulos. Más importante aún, la misma estrategia puede ampliarse para incluir no sólo productos entregables sino tambien las sucesivas versiones intermedias creadas en una estrategia de construcción incremental. El producto se desarrolla entonces a través de sus etapas intermedias con un mínimo retroceso.

El enfoque de Microsoft "Construir Cada Noche"

James McCarthy me describió un proceso de producto usado por su equipo y otros en Microsoft. Es un desarrollo incremental llevado a una conclusión lógica. Afirma,

> *Después de distribuir la primera versión, distribuiremos versiones posteriores que añadan más funcionalidad a un producto existente en uso. Por qué el proceso inicial de construcción debe ser diferente? Empezando por el momento de nuestro primer hito [donde la marcha hacia la primera distribución tiene tres hitos intermedios] reconstruimos el sistema en desarrollo cada noche [y ejecutamos los casos de prueba]. El ciclo de construcción se convierte en el motor del proyecto. Cada día, uno o más equipos de programadores-probadores verifican los módulos con nuevas funcionalidades. Después de cada contrucción, tenemos un sistema en ejecución. Si la construcción fracasa, paramos todo el proceso hasta que se encuentre el problema y se arregle. En todo momento todos en el equipo conocen el estado.*
>
> *Es muy difícil. Hay que dedicar muchos recursos, pero es un proceso disciplinado, un proceso monitorizado y conocido. Esto le da credibilidad al equipo. Tu credibilidad determina tu moral y tu estado emocional.*

Los constructores de software en otras organizaciones están sorprendidos, incluso conmocionados, por este proceso. Se dice, "He hecho un ensayo para construir cada semana, pero creo que sería demasiado trabajo construir cada noche." Y eso puede ser cierto. Por ejemplo, Bell Northern Research reconstruye su sistema de 12 millones de líneas de código cada semana.

Construcción Incremental y Creación Rápida de Prototipos

Puesto que un proceso de desarrollo incremental nos permite pruebas tempranas con usuarios reales, ¿Cuál es la diferencia entre eso y la creación de prototipos rápidos? Me parece que ambos están relacionados pero separados. Se puede tener uno sin el otro.

Harel define útilmente un prototipo como

> *[Una versión de un programa que] refleja sólo las decisiones de diseño tomadas en el proceso de preparación del modelo conceptual, y no las decisiones motivadas por cuestiones de implementación.*[12]

Es posible construir un prototipo que no es en absoluto parte de un producto en desarrollo para su distribución. Por ejemplo, se puede construir un prototipo de interfaz que no tenga ninguna función real en el programa detrás de él, únicamente la máquina de estados finito que lo hace parecer que va atravensando sus etapas. Incluso se pueden crear prototipos y probar interfaces a través de la técnica del Mago de Oz, con un humano escondido simulando las respuestas del sistema. Tal creación de prototipos puede ser muy útil para obtener retroalimentación temprana del usuario, aunque está bastante separado de las pruebas de un producto desarrollado para su distribución.

De forma similar, los implementadores bien pueden emprender la construcción de una rebanada vertical de un producto, en la que se contruya por completo un conjunto de funcionalidades muy limitadas, de manera que permita iluminar tempranamente los lugares donde puede acechar la serpiente del rendimiento. ¿Cuál es la diferencia entre el primer hito construido del proceso de Microsoft y un prototipo rápido? La funcionalidad. El producto del primer hito puede no tener la funcionalidad suficiente que sea de interés para nadie; el producto distribuible está definido como tal por su completitud en suministrar un conjunto útil de funcionalidades, y por su calidad, la creencia de que funciona de forma robusta.

Parnas Estaba en lo Correcto, y Yo Estaba Equivocado acerca del ocultamiento de Información

En el Capítulo 7 contrasto dos enfoques a la pregunta de cuánto se debe permitir o animar conocer a cada miembro del equipo acerca del diseño y codificación del otro. En el proyecto Operating System/360, decidimos que *todos* los programadores deberían ver *todo* el material – i.e. que todo programador debería tener una copia del cuaderno de trabajo del proyecto, que llegó a sumar más de 10,000 páginas. Harlam Mills ha argumentado persuasivamente que "la programación debería ser un proceso público," que la exposición de todo el trabajo a la vista de todos ayuda al control de calidad, ya sea por presiones de sus pares para hacer bien las cosas como por los compañeros que realmente detectan defectos y errores.

Esta visión contrasta marcadamente con las enseñanzas de David Parnas de que los módulos de código deben ser encapsulados con interfaces bien definidas, y que el interior de dicho módulo debería ser propiedad privada

de su programador, imperceptible desde el exterior. Los programadores son más eficientes si se los resguarda de, no si se los expone a, las entrañas de los módulos que no son de su propiedad.[13]

En el capítulo 7 rechacé el concepto de Parnas como una "receta para el desastre." Parnas estaba en lo correcto, y yo estaba equivocado. Ahora estoy convencido de que ocultar la información, hoy en día a menudo incorporada en la programación orientada a objetos, es la única manera de elevar el nivel del diseño de software.

Asimismo se puede fracasar con cualquier técnica. La técnica de Mill asegura que los programadores pueden conocer la semantica detallada de las interfaces en las que trabajan sabiendo lo que hay al otro lado. Ocultar esa semántica conduce a errores del sistema. Por otro lado, la técnica de Parnas es robusta bajo el cambio y es más apropiada en una filosofía de diseño para el cambio.

El Capítulo 16 arguye lo siguiente:

- La mayor parte del progreso anterior en la productividad del software ha venido de la eliminación de dificultades no inherentes, como los lenguajes de máquina incómodos y el lento tiempo de respuesta del proceso por lotes.
- No existen muchas más de estas elecciones cómodas.
- Un progreso radical tendrá que venir de atacar las dificultades esenciales de la creación de construcciones conceptuales complejas.

La manera más obvia de hacer esto es reconocer que los programas están hechos de pedazos conceptuales mucho más grandes que las declaraciones individuales de los lenguajes de alto nivel – subrutinas, o módulos, o clases. Si podemos limitar el diseño y la construcción tal que solo hagamos el montaje y la parametrización de tales pedazos a partir de colecciones prefabricadas, habremos elevado radicalmente el nivel conceptual, y eliminado la vasta cantidad de trabajo y las copiosas oportunidades de error que habitan en el nivel de las declaraciones individuales.

La definicion de Parnas del ocultamiento de información de módulos es el primer paso publicado en ese programa de investigación crucialmente importante, y es el antepasado intelectual de la programación orientada a objetos. Parnas define un módulo como una entidad de software con su propio modelo de datos y su propio conjunto de operaciones. Sus datos solo pueden

ser accedidos mediante sus propias operaciones. El segundo paso fue una contribucion de varios pensadores: la promoción del módulo de Parnas en un *tipo de datos abstracto*, a partir del cual se pueden derivar muchos objetos. El tipo abstracto de datos proporciona una manera uniforme de pensar y especificar las interfaces del módulo, y una disciplina de acceso que es fácil de aplicar.

El tercer paso de la programacion orientada a objetos, introduce el poderoso concepto de *herencia*, por el cual las clases (tipos de datos) toman por omision atributos específicos de sus antepasados en la jerarquía de clases.[14] Lo que más esperamos ganar de la programacion orientada a objetos se deriva de hecho del primer paso, el encapsulamiento de módulos, más la idea de bibliotecas prefabricadas de módulos o clases *que están diseñados y probados para su reuso.* Mucha gente ha elegido ignorar el hecho de que tales módulo no son sólo programas, sino que son productos de programas en el sentido discutido en el Capitulo 1. Algunas personas esperan en vano un reuso significativo de módulos sin pagar el costo inicial de la construcción de módulos de productos de calidad - generalizados, robustos, probados y documentados. La programación orientada a objetos y el reuso se han discutido en los Capitulos 16 y 17.

¿Cuán Mítico Es el Hombre-Mes? Modelo y Datos de Boehm

A través de los años se han realizado muchos estudios cuantitativos sobre la productividad del software y de los factores que lo afectan, especialmente los compromisos entre el personal del proyecto y el calendario.

El estudio más amplio es uno realizado por Barry Boehm de unos 63 proyectos de software, en su mayoria aeroespaciales, con unos 25 en TRW. Su libro *Software Engineering Economics* contiene no solo los resultados sino un conjunto útil de modelos de costo de integralidad progresiva. Mientras que los coeficientes en los modelos son con seguridad diferentes para el software comercial común y para el software aeroespacial construido según los estándares del gobierno, sin embargo, sus modelos están respaldados por una inmensa cantidad de datos. Creo que el libro será todo un clásico útil dentro de una generación.

Sus resultados confirman de forma sólida las afirmaciones de *El MH-M* de que el compromiso entre personas y meses está lejos de ser lineal, que el

hombre-mes es en efecto un mito como medida de productividad. En especial, encuentra que:[15]

- Existe un tiempo en el calendario cuyo costo para la primera entrega es óptimo, $t = 2.5 \times (HM)^{1/3}$. Es decir, el tiempo óptimo en meses es la raíz cúbica del esfuerzo esperado en hombre-meses, una cifra derivada de la estimación del tamaño y otros factores en su modelo. Un corolario es una curva óptima de personal.

- La curva de costo aumenta lentamente a medida que el calendario programado se alarga más allá del óptimo. Las personas con más tiempo tardan más.

- La curva del costo aumenta bruscamente a medida que el calendario programado se acorta más que el óptimo.

- *¡Casi ningún proyecto tendrá éxito en menos de 3/4 del calendario óptimo, independientemente de la cantidad de gente involucrada!* Este resultado notable brinda al gestor de software munición sólida cuando la gestión de mayor nivel exige compromisos de calendario imposibles.

¿Cuán verdadera es la Ley de Brooks? Incluso se han realizado cuidadosos estudios que evalúan la verdad de la Ley de Brooks (intencionalmente simplistas), que añadir fuerza de trabajo a un proyecto de software retrasado lo retrasa aún más. El mejor estudio es el de Abdel-Hamid y Madnick, en su ambicioso y valioso libro de 1991, *Software Project Dynamics: An integrated approach*[16]. El libro desarrolla un modelo cuantitativo de la dinámica del proyecto. Su capítulo sobre de la Ley de Brooks proporciona una visión más detallada acerca de lo que sucede bajo varios supuestos en cuanto a qué mano de obra se añade y cuándo. Para investigar esto, los autores extienden su propio modelo cuidadoso de un proyecto de aplicaciones de tamaño medio asumiendo que las nuevas personas tienen una curva de aprendizaje y son responsables del trabajo extra de comunicación y capacitación. Concluyen que "Añadir más gente a un proyecto retrasado siempre lo hace más costoso, pero no *siempre* hace que se complete más tarde [las cursivas son de ellos]." En particular, añadir mano de obra extra al comienzo del calendario es una

maniobra más segura que añadirla después, puesto que la nuevas personas siempre tienen un efecto negativo inmediato, que tarda semanas en compensarse.

Stutzke desarrolla un modelo más simple para efectuar una investigacion similar, con un resultado similar.[17] Desarrolla un análisis detallado del proceso y los costos de asimilación de los nuevos trabajadores, incluyendo explícitament la distracción de sus mentores de la propia tarea del proyecto. Prueba su modelo en contra de un proyecto real en la cual la mano de obra se duplicó con éxito y se llevó a cabo el calendario original, después de un retraso a mitad del proyecto. Considera alternativas para añadir más progamadores, especialmente hora extras. Lo más valioso son sus muchos consejos prácticos sobre cómo se debe añadir, capacitar y apoyar con herramientas a los nuevos trabajadores, etc, para minimizar los efectos perjudiciales de su incorporación. En especial es digno de mención su comentario de que la gente añadida tardíamente a un proyecto de desarrollo deben ser miembros de un equipo dispuesto a colaborar y trabajar dentro del proceso y no intentar alterar o mejorar el proceso mismo!

Stutzke cree que la carga adicional de comunicación en un proyecto más grande es un efecto de segundo orden y no lo modela. No está claro si Abdel-Hamid y Madnick lo toman en cuenta y cómo. Ningún modelo toma en cuenta el hecho de que el trabajo debe ser repartido, un proceso que a menudo he encontrado que no es trivial.

La afirmación "escandalosamente simplificada" de la ley de Brooks se hace más útil gracias a estos estudios minuciosos de las competencias apropiadas. En general, mantengo esa afirmación llana como la mejor aproximación de orden cero a la verdad, una regla general de advertencia a los gestores en contra de realizar ciegamente la corrección instintiva a un proyecto retrasado.

La Gente Es todo (Bueno, Casi Todo)

Algunos lectores han encontrado curioso que *El MH-M* dedique la mayor parte de los ensayos a los aspectos de gestión de la ingeniería de software, más que a las cuestiones técnicas. Este sesgo se debió en parte a la naturaleza de mi papel en el sistema operativo IBM Operating System/360 (ahora MVS/370). Más fundamentalmente, surgió de mi convicción de que la calidad de la gente

que participa en un proyecto, así como su organización y gestión, son factores mucho más importantes para el éxito que las herramientas que usan o los enfoques técnicos que adoptan.

Investigaciones posteriores han respaldado esa convicción. El modelo COCOMO de Boehm establece que la calidad del equipo, es con mucho, el factor más importante para su éxito, de hecho cuatro veces más fuerte que el siguiente factor más importante. La mayor parte de la investigación académica acerca de la ingeniería de software se ha concentrado en las herramientas. Admiro y codicio las herramientas afiladas. Sin embargo, es alentador ver trabajos de investigación en curso acerca del cuidado, el desarrollo y la alimentación de la gente, y acerca de la dinámica de la gestión del software.

Peopleware. Una gran contribucion durante los años recientes ha sido el libro de DeMarco y Lister de 1987, *Peopleware: Productive Project and Teams*. Su tesis subyacente es que "Los principales problemas de nuestro trabajo no son tanto de naturaleza *tecnológica* como *sociológica*." Abunda en gemas tales como, "La función del gestor no es hacer que la gente trabaje, sino hacer posible que la gente trabaje." Trata con cuestiones mundanas tales como el espacio, los muebles, las comida en equipo. DeMarco y Lister proporcionan datos reales de su Coding War Games que muestran una correlación sorprendente entre el rendimiento de los programadores de una misma organización, y entre las características del lugar de trabajo y los niveles de productividad y de defectos.

> *El espacio de los con mayor rendimiento es más silencioso, más privado, está mejor protegido contra las interrupciones, y hay más . . . ¿Realmente te importa? . . . ya sea que la tranquilidad, el espacio y la privacidad ayuden a su gente a realizar mejor su trabajo o [alternativamente] ¿Te ayuden a atraer y a conservar a la mejor gente?*[18]

Recomiendo de corazón el libro a todos mis lectores.

Proyectos en movimiento. Demarco y Lister prestan una considerable atención a la *fusión* de equipo, una propiedad intangible pero vital. Pienso que el hecho de que la gestión pase por alto la fusión es lo que explica la buena disposición que he observado en las empresas con múltiples ubicaciones para trasladar un proyecto de un laboratorio a otro.

Mi experiencia y observación se limitan a media docena de movimientos quizá. Nunca he visto uno exitoso. Uno puede mover misiones con éxito. Pero en cada caso de intentos de trasladar proyectos, el nuevo equipo de hecho empezaba de nuevo, a pesar de tener una buena documentación, algunos diseños bien avanzados y algunas personas del equipo emisor. Creo que es la ruptura de la fusión del antiguo equipo lo que aborta el producto embrionario y provoca su reinicio.

El Poder para Ceder el Poder

Si uno cree, como he argumentado en mucho lugares en este libro, que la creatividad proviene de individuos y no de estructuras o procesos, entonces una pregunta central que encara la gestión de software es cómo diseñar estructuras y procesos para mejorar, en lugar de inhibir, la creatividad e iniciativa. Afortunadamente, este problema no es propio de las organizaciones de software, y grandes pensadores han trabajado en esto. E.F. Schumacher, en su clásico libro, *Lo Pequeño es Hermoso*[†], propone una teoría de la organización empresarial para maximizar la creatividad y el disfrute de los trabajadores. Como su primer principio elige el "Principio de la Función Subsidiaria" de la Encíclica *Quadragesimo Anno* del Papa Pío 11:

> *es injusto, y al mismo tiempo de grave perjuicio y perturbación para el recto orden social, confiar a una sociedad mayor y más elevada lo que comunidades menores e inferiores pueden hacer y procurar. Toda acción de la sociedad debe, por su naturaleza, prestar auxilio a los miembros del cuerpo social, más nunca absorberlos y destruirlos . . . Los que gobiernan tengan bien entendido esto: que cuando más vigorosamente reine el orden jerárquico entre las diversas asociaciones, quedando en pie este principio de la función subsidiaria del Estado, tanto más firme será la autoridad y el poder social, y tanto más próspera y feliz la condición del Estado.*[19]

Schumacher pasa a la interpretación:

> *El Principio de Función Subsidiria nos enseña que el centro ganará en autoridad y eficacia si la libertad y responsabilidad de las formaciones menores son*

[†]Editorial Hermann Blume, 1978

> *preservadas cuidadosamente, con el resultado de que la organización como un todo será "más feliz y más próspera."*
>
> *¿Cómo se puede conseguir tal estructura? . . . La gran organización constará de muchas unidades semiautónomas, a las que podemos llamar* quasi-empresas. *Cada una de ellas tendrá una gran cantidad de libertad, para dar la mayor oportunidad posible a la creatividad y el* emprendimiento . . . *Cada cuasi-empresa debe tener una cuenta de ganancias y pérdidas y un balance general.*[20]

Entre los desarrollos más emocionantes en ingeniería de software están las primeras etapas de la puesta en práctica de estas ideas organizativas. Primero, la revolución de las microcomputadoras creó una nueva industria de software con cientos de emprendimientos, todas estas empresas pequeñas y marcadas por el entusiasmo, la libertad y la creatividad. La industria está cambiando ahora, ya que muchas compañías pequeñas están siendo absorvidas por otras más grandes. Queda por ver si los grandes compradores entenderán la importancia de preservar la creatividad de la pequeñez.

De forma más notable, la alta gestión en algunas empresas grandes han empezado a delegar el poder a equipos de proyectos de software individuales, lo que los ha llevado a aproximarse a las cuasi empresas de Shumacher en estructura y responsabilidad. Están sorprendido y encantados con los resultados.

Jim McCarthy de Microsoft me ha descrito su experiencia al emancipar sus equipos.

> *Cada equipo en particular (30-40 personas) es dueño de su propio conjunto de características, calendario e incluso su proceso de cómo definir, construir y distribuir. El equipo está compuesto hasta por cuatro o cinco especialidades, incluyendo la construcción, las pruebas y la redacción. El equipo resuelve las disputas; los jefes no. No puedo enfatizar lo suficiente acerca de la importancia del empoderamiento, de que el equipo sea reponsable ante sí mismo por su éxito.*

Earl Wheeler, jefe jubilado del software de negocios de IBM, contó su experiencia en llevar a cabo la delegación descendente del poder durante mucho

tiempo centralizado en las administraciones de departamentos de IBM:

> *El impulso clave [de los últimos años] fue delegar el poder. ¡Fue como magia! Mejoró la calidad, la productividad y la moral. Tenemos equipos pequeños sin control central. Los equipos son dueños del proceso, pero tienen que tener uno. Tienen muchos procesos diferentes. Son dueños del calendario, pero tienen la presión del mercado. Esta presión hace que busquen herramientas por su cuenta.*

Las pláticas con miembros individuales del equipo, por supuesto, muestran tanto una apreciación del poder y la libertad que se delega, como una estimación algo más conservadora de a cuánto control se renuncia realmente. No obstante, la delegación lograda es claramente un paso en la dirección correcta. Produce exactamente los beneficios que Pío 11 predijo: el centro gana en autoridad real al delegar poder, y la organización en su conjunto es más feliz y más próspera.

¿Cuál es la Nueva Mayor Sorpresa? Millones de Computadoras

Todos los gurús del software con los que he hablado admiten haber sido sorprendidos por la revolución de las microcomputadoras y su consecuencia, la industria del software comercial. Este es, sin duda, el cambio crucial de las dos décadas desde *El MH-M*, tiene muchas implicaciones para la ingeniería de software.

La revolución de las microcomputadoras ha cambiado la forma en que todos usan las computadoras. Schumacher planteó el reto hace más de 20 años:

> *¿Qué es lo que realmente requerimos de los científicos y tecnólogos? Debería responder: Necesitamos métodos y equipo que sean*
>
> - *suficientemente baratos tal que sean accesibles virtualmente por todos;*
> - *adecuados para aplicaciones de pequeña escala; y*
> - *compatibles con las nececidades creativas de la persona.*[21]

Estas son exactamente las maravillosas propiedades que la revolución de las microcomputadoras ha traído a la industria de las computadoras y a sus usuarios, hoy al público en general. El estadounidense promedio ahora puede costearse no sólo una computadora propia, sino una paquete de software que hace 20 años habría costado el salario de un rey. Vale la pena contemplar cada una de las metas de Schumacher; vale la pena saborear el grado en que cada una de ellas ha sido conseguida, en especial la última. Área tras área, los nuevos medios de autoexpresión son accesibles tanto por la gente común como por los profesionales.

En parte, como en la creación del software la mejora en otros campos proviene de la eliminación de las dificultades accidentales. La edición de manuscritos solían ser rígidos accidentalmente por el tiempo y el costo de reeditarlos para incorporar cambios. En una obra de 300 páginas, uno podía volver a escribir cada tres a seis meses, pero en medio, uno seguía poniendo marcas en el manuscrito. No se podía evaluar fácilmente qué habían hecho los cambios al flujo de la lógica y al ritmo de las palabras. Hoy en día, los manuscritos se han vuelto maravillosamente fluidos. [22]

La computadora ha aportado una fluidez similar a muchos otros medios: dibujos artísticos, planos de edificios, diseños mecánicos, composiciones musicales, fotografías, secuencias de video, presentaciones de diapositivas, trabajos multimedia e incluso hojas de cálculo. En cada caso, el método de producción manual requería volver a copiar las partes voluminosas no modificadas para ver los cambios de contexto. Ahora disfrutamos para cada medio los mismos beneficios que el tiempo compartido trajo a la creación de software – la capacidad de revisar y evaluar instantáneamente el efecto sin perder el tren de pensamiento.

La creatividad también se ve reforzada por las nuevas y flexibles herramientas auxiliares. Para la producción de prosa, por ejemplo, ahora contamos con correctores ortográficos, correctores gramaticales, asesores de estilo, sistemas bibliográficos y la notable capacidad de ver páginas formateadas simultáneamente en el diseño final. Todavía no apreciamos lo que las enciclopedias instantáneas o los infinitos recursos de la World-Wide Web significarán para la investigación improvisada de un escritor.

Lo más importante es que la nueva fluidez de los medios de comunicación facilita la exploración de muchas alternativas radicalmente diferentes cuando

una obra creativa acaba de tomar forma. He aquí otro caso en el que un orden de magnitud en un parámetro cuantitativo, en este caso el tiempo de cambio, produce una diferencia cualitativa en cómo se realiza una tarea.

Las herramientas de dibujo permiten a los diseñadores de edificios explorar muchas más opciones por hora de inversión creativa. La conexión de computadoras a sintetizadores, con software para generar o reproducir partituras automáticamente, facilita mucho la captura de los garabatos del teclado. La manipulación de fotografías digitales, como con Adobe Photoshop, permite realizar experimentos de minutos de duración que durarían horas en un cuarto oscuro. Las hojas de cálculo permiten la fácil exploración de docenas de escenarios alternativos de "qué pasaría si".

Por último, la ubicuidad de la computadora personal ha permitido la aparición de medios creativos totalmente nuevos. Los hipertextos, propuestos por Vannevar Bush en 1945, sólo son prácticos con las computadoras. Las presentaciones y experiencias multimedia eran un gran problema -demasiado problemáticos- antes de la disponibilidad de la computadora personal y el software abundante y barato. Los sistemas de ambientes virtuales, que todavía no son baratos ni omnipresentes, lo serán, y serán otro medio creativo más.

La revolución de las microcomputadoras ha cambiado la forma en la que todos construyen software. Los procesos de software de la década de los 70's se han visto alterados por la revolución del microprocesador y los avances tecnológicos que lo posibilitaron. Muchas de las dificultades accidentales de esos procesos de construcción de software han sido eliminadas. Las veloces computadoras individuales son ahora las herramientas rutinarias del desarrollador de software, por lo que el tiempo de respuesta es un concepto casi obsoleto. La computadora personal de hoy no sólo es más rápida que la super computadora de 1960, es más rápida que la estación de trabajo de Unix de 1985. Todo esto significa que la compilación es rápida incluso en los equipos más modestos, y las grandes memorias han eliminado las esperas para el enlace en disco. Las grandes memorias también hacen que sea lógico mantener tablas de símbolos en memoria junto al código objeto, de modo que la depuración de alto nivel sin recompilación es algo rutinario.

En los últimos 20 años, hemos conseguido casi por completo el uso del

tiempo compartido como metodología para construir software. En 1975, el tiempo compartido acababa de reemplazar a la computación por lotes como la técnica más común. La red se utilizó para dar acceso al creador de software tanto a los archivos compartidos como a un poderoso motor compartido de compilación, enlace y pruebas. Hoy en día, la estación de trabajo personal proporciona el motor de cómputo, y la red principalmente da acceso compartido a los archivos que son el producto del trabajo de desarrollo del equipo. Los sistemas cliente-servidor hacen que el acceso compartido al registro, a la construcción y a la aplicación de casos de prueba sea un proceso diferente y más sencillo.

Se han producido avances similares en las interfaces de usuario. La interfaz VIMP provee una edición más práctica de programas como también de textos en lengua Inglesa. La pantalla de 24 líneas y 72 columnas ha sido reemplazada por pantallas de una página completa o incluso de dos páginas, asi que los programadores pueden ver mucho más contexto para los cambios que realizan.

Toda una Nueva Industria de Software – El Software Comercial

Junto a la clásica industria del software ha explotado otra. Las ventas de unidades de productos ascienden a cientos de miles e incluso millones. Se pueden adquirir paquetes de software completos por menos de lo que cuesta financiar un día de programador. Las dos industrias son diferentes en muchos aspectos y coexisten

La industria clásica del software. En 1975, la industria del software tenía varios componentes identificables y un tanto diferentes, todos ellos todavía existen actualmente:

- Los vendedores de computadoras, que proveen sistemas operativos, compiladores y programas de soporte para sus productos.
- Los usuarios de aplicaciones, como las tiendas de SGI de programas de soporte, bancos, compañías de seguros y agencias gubernamentales, que construyen paquetes de aplicaciones para su propio uso.

- Los constructores de aplicaciones personalizadas, que contratan para la construcción de paquetes de software propietario para los usuarios. Mucho de estos contratistas se especializan en aplicaciones de defensa, donde los requerimientos, estándares y procedimientos de comercialización son especiales.
- Los desarrolladores de paquetes comerciales, que en ese momento desarrollaron principalmente grandes aplicaciones para mercados especializados, tales como paquetes de análisis estadístico y sistemas CAD.

Tom DeMarco señala la fragmentacion de la clásica industria del software, especialmente el componente de aplicaciones de usuario:

> *Lo que no esperaba: el campo se ha dividido en nichos. La forma en que se hace algo es mucho más una función del nicho que el uso de métodos generales de análisis de sistemas, lenguajes generales y técnicas generales de pruebas. Ada fue el último de los lenguajes de propósito general, y se ha convertido en un lenguaje de nicho.*

En el nicho de aplicaciones comerciales de rutina, los lenguajes de cuarta generación han hecho contribuciones muy importantes. Boehm dice, "Las 4GLs - cuatro generaciones de lenguajes- más exitosas son el resultado de que alguien codifica una parte de un dominio de aplicación en términos de opciones y parámetros". El más ubicuo de estas 4GLs son los generadores de aplicaciones y los paquetes combinados de comunicaciones de bases de datos con lenguajes de consulta.

Los mundos de los sistemas operativos se han fusionado. En 1975, los sistemas operativos abundaban: cada proveedor de hardware tenía al menos un sistema operativo propietario por línea de productos; muchos tenían dos. ¡Cuán diferentes son las cosas hoy en día! Los sistemas abiertos son la consigna, y solo existen cinco entornos de sistemas operativos importantes en los que las personas comercializan paquetes de aplicaciones (en orden cronológico)

- Los entornos MVS y VM de IBM

- El entorno VMS de DEC
- El entorno Unix, de un sabor u otro
- El entorno ya sea DOS, OS-2 o Windows de la PC de IBM
- El entorno de Apple Macintosh

La industria del software comercial. Para el desarrollador en la industria del software comercial, la economía es totalmente diferente a la de la industria clásica: el costo de desarrollo se divide por grandes cantidades; los costos de embalaje y comercialización van en aumento. En el clásico desarrollo de aplicaciones internas de la industria, el calendario y los detalles de la funcionalidad eran negociables, el costo de desarrollo podía no serlo; en el mercado abierto y ferozmente competitivo, el calendario y funcionalidad dominan bastante el costo de desarrollo.

Como era de esperar, las marcadas diferencias económicas han dado lugar a culturas de programación bastante diferentes. La industria clásica tendía a estar dominada por grandes empresas con estilos de gestión y culturas de trabajo establecidos. Por otro lado, la industria del software comercial comenzó como cientos de emprendimientos, con un enfoque independiente y ferozmente concentrados en la realización del trabajo en lugar de en el proceso. En este clima, siempre ha habido un reconocimiento mucho mayor al talento del programador individual, una conciencia implícita de que los grandes diseñadores producen grandes diseños. La cultura del emprendimiento tiene la capacidad de recompensar a los artistas estelares en proporción a sus contribuciones; en la clásica industria del software, la sociología de las corporaciones y sus planes de gestión salarial siempre han hecho esto difícil. No es de extrañar que muchas de las estrellas de la nueva generación se hayan decantado por la industria del software comercial.

Compre y Construya – Paquetes de Software Comerciales Como Componentes

Se va a lograr la robustez y la productividad del software radicalmente mejores solo subiendo un nivel, y creando programas por medio de la composición de

módulos u objetos. Una tendencia especialemente promisoria es el uso de paquetes de mercado masivo como plataformas sobre las que se contruyen productos más ricos y personalizados. Se construye un sistema de seguimiento de camiones sobre una base de datos comercial y un paquete de comunicaciones; lo mismo que un sistema de información para estudiantes. Los anuncios clasificados en las revistas de computación ofrecen cientos de pilas de HyperCard y plantillas personalizadas para Excel, docenas de funciones especiales en Pascal para MiniCad o funciones en AutoLisp para Autocad.

Metaprogramación A la construcción de pilas de HiperCard, de plantillas de Excel o de funciones de MiniCad a veces se la llama *metaprogramación*, la construcción de una nueva capa que personaliza la funcionalidad para un subconjunto de usuarios de un paquete. El concepto de metaprogramación no es nuevo, sólo resurgió y cambió de nombre. A pincipios de 1960, los vendedores de computadoras y muchas grandes tiendas de sistemas de gestión información (SGI) tenían pequeños grupos de especialistas que creaban lenguajes de programación de una aplicación completa a partir de macros en lenguaje ensamblador. La tienda de SGI de Kodak Eastman tenía un lenguaje de aplicación propio definido sobre el macroensamblador IBM 7080. Igualmente, con el Método de Acceso a las Telecomunicaciones en Cola de la IBM OS/360, se podían leer muchas páginas de un programa de telecomunicaciones aparentemente en lenguaje ensamblador antes de encontrar una instrucción en lenguaje de máquina. Ahora, los fragmentos ofrecidos por el metaprogramador son muchas veces más grandes que esos macros. Este desarrollo de mercados secundarios es muy alentador – mientras hemos estado esperando ver que se desarrolle un mercado efectivo de clases en C++, un mercado de metaprogramas reusables ha crecido sin ser notado.

Esto realmente ataca la esencia. Debido a que el fenómeno de la contrucción basada en paquetes no afecta actualmente al programador promedio de SGI, aún no es muy visible para el área de ingeniería de software. Sin embargo, crecerá rapidamente, porque ataca la esencia de la creación de contrucciones conceptuales. El paquete comercial proporciona un gran módulo de funcionalidad, con una interfaz compleja pero adecuada, y su estructura conceptual interna no tiene que ser diseñada en absoluto. Los productos de

software de elevada funcionalidad tales como Excel o 4th Dimension son en efecto grandes módulos, pero sirven como módulos conocidos, documentados y probados con los que se pueden construir sistemas personalizados. Los creadores de aplicaciones del siguiente nivel obtienen una funcionalidad rica, un tiempo de desarrollo más corto, un componente probado, mejor documentación y un costo radicalmente inferior.

Por supuesto, la dificultad es que el paquete de software comercial está diseñado como una entidad autónoma cuyas funcionalidades e interfaces los metaprogamdores no pueden cambiar. Además, y lo que es más grave, los constructores de paquetes comerciales parecen tener pocos incentivos para hacer que sus productos sean adecuados como módulos en un sistema mayor. Creo que esa percepción está errada, que existe un mercado sin explotar en el suministro de paquetes diseñados para facilitar el trabajo del metaprogramador.

Entonces, ¿Qué se necesita? Podemos identificar cuatro niveles de usuarios de paquetes comerciales:

- El usuario tal cual, que maneja la aplicación de forma directa, satisfecho con las funcionalidades y la interfaz que los diseñadores proveen.

- El metaprogramador, que construye plantillas o funciones en el nivel superior de una aplicación, usa la interfaz disponible, principalmente para ahorrarle trabajo al usuario final.

- El escritor de funciones externas, quien codifica manualmente las funciones que se añaden a una aplicación. Se trata básicamente de nuevas primitivas del lenguaje de aplicación que llaman a módulos de código separados escritos en un lenguaje de propósito general. Se necesita la capacidad de interconectar estas nuevas funciones a la aplicación como comandos interceptados, como retrollamadas o como funciones sobrecargadas.

- El metaprogramador que usa una, o sobre todo varias aplicaciones como componentes en un sistema mayor. Este es el usuario cuyas necesidades están actualmente mal cumplidas. Este es también el uso que promete

beneficios efectivos substanciales en la construcción de nuevas aplicaciones.

Para este último usuario, una aplicación comercial necesita una interfaz documentada adicional, la interfaz de la metaprogramación (IMP). Necesita varias competencias. Primero, el metaprograma necesita tener el mando de un montaje de aplicaciones, mientras que normalmente cada aplicación asume que está al mando. El montaje debe controlar la interfaz de usuario, que normalmente asume la aplicación que está haciendo. El montaje debe ser capaz de invocar cualquier función de aplicación como si su cadena de comando hubiera venido del usuario. Debe recibir la salida de la aplicación como si fuera la pantalla, excepto que necesita que la salida sea analizada sintácticamente en unidades lógicas de tipos de datos adecuados, en lugar de la cadena de texto que se habría desplegado. Algunas aplicaciones, tales como Foxpro, tienen agujeros de gusano que nos permiten pasar una cadena de comando de entrada, aunque la informacion que se obtiene de regreso es escasa y sin un análisis sintáctico. El agujero de gusano es una solución *ad hoc* para una necesidad que exige una solución diseñada de forma general.

Es poderoso tener un lenguaje de archivo de comandos para controlar las interacciones entre el conjunto de aplicaciones. Unix fue el primero en proporcionar este tipo de funcionalidad, con sus tuberías y su formato de archivo de cadena ASCII estándar. Actualmente AppleScript es un ejemplo bastante bueno.

El Estado Actual y el Futuro de la Ingeniería de Software

Una vez le pedí a Jim Farrel, presidente del Departamento de Ingeniería Química de la Universidad Estatal de Carolina del Norte, que relatara la historia de la ingeniería química, para distinguirla de la química. En seguida dió un maravilloso relato improvisado de una hora de duración, empezando con la existencia, desde la antigüedad, de muchos procesos diferentes de producción de muchos productos, desde el acero hasta el pan y el perfume. Me contó cómo el Profesor Arthur D. Little fundó un Departamento de Química Industrial en el MIT en 1928, para encontrar, desarrollar y enseñar una base común de técnicas compartidas por todos los procesos. Primero, fueron las reglas generales, luego los nomogramas empíricos, luego las fórmulas para el diseño de compo-

nentes particulares, después los modelos matemáticos para la transferencia de calor, la transferencia de masa, la transferencia del momentum en recipientes individuales.

A medida que se desarrollaba el relato de Ferrel, me sorprendieron los muchos paralelismos entre el desarrollo de la ingeniería química y de la ingeniería de software, casi exactamente cincuenta años después. Parnas me reprocha por haber escrito acerca de la *ingeniería de software*. Él contrasta la disciplina de software con la ingeniería eléctrica y siente que es una presunción llamar a lo que hacemos ingeniería. Puede que tenga razón en que el campo nunca se convertirá en una disciplina de la ingeniería con una base matemática tan precisa y completa como la que tiene la ingeniería eléctrica. Después de todo, la ingeniería de software, como la ingeniería química, está involucrada con los problemas no lineales de la ampliación a procesos a escala industrial, y al igual que la ingeniería industrial, está permanentemente confundida por las complejidades del comportamiento humano.

Sin embargo, el curso y el ritmo de desarrollo de la ingeniería química me lleva a creer que la ingeniería de software a la edad de 27 años puede no estar desesperada sino simplemente inmadura, como lo estaba la ingeniería química en 1945. Fue solo después de la segunda guerra mundial que los ingenieros químicos se ocuparon realmente del comportamiento de los sistemas de flujo contínuo interconectados en lazo cerrado.

Los asuntos particulares de la ingeniería de software son hoy en día precisamente las expuestas en el Capítulo 1.

- Cómo diseñar y construir un conjunto de programas dentro de un *sistema*.
- Cómo diseñar y construir un programa o un sistema dentro de un *producto* robusto, probado, documentado y respaldado
- Cómo mantener el control intelectual sobre la *complejidad* en grandes dosis.

El pozo de brea de la ingeniería de software continuará siendo pegajoso por un largo tiempo. Uno puede esperar que la raza humana cotinúe intentando sistemas dentro o fuera de nuestro alcance; los sistemas de software son quizás las más intrincadas obras del hombre. Ese complejo oficio exigirá nuestro

continuo desarrollo de la disciplina, nuestro aprendizaje para componer en unidades mayores, nuestro mejor uso de nuevas herramientas, nuestra mejor adaptación de métodos comprobados de gestión de ingeniería, una aplicación flexible del sentido común, y una humildad otorgada por dios para reconocer nuestra falibilidad y limitaciones.

Epílogo
Cincuenta Años de Asombro, Emoción, y Placer

Todavía perdura en mi mente el asombro y deleite con el cual – en ese entonces de 13 años – leí el relato del 7 de Agosto de 1944, dedicado a la computadora Harvard Mark I, una maravilla electromecánica cuyo arquitecto era Howard Aiken y los ingenieros Clair Lake, Benjamin Durfee y Francis Hamilton los diseñadores de la implementación. Igualmente me asombró la lectura del artículo "That We May Think" de Vannevar Bush en el *Atlantic Monthly* de Abril de 1945, en el cual proponía organizar el conocimiento como una gran red de hipertexto dotando a las máquinas usuarios a seguir rutas existentes como de trazar nuevas rutas de asociaciones.

Mi pasión por las computadoras recibió otro gran impulso en 1952, cuando un trabajo de verano en IBM en Endicott, New York, me dió la experiencia práctica en la programación de la IBM 604 y la instrucción formal en la programación de la IBM 701, su primera máquina con programa almacenado. La escuela de posgrado bajo Aiken e Iverson en Harvard hizo el sueño de mi carrera una realidad, y fuí atrapado de por vida. A sólo una fracción de la raza humana dios le da el privilegio de ganarse el pan haciendo lo que uno hubiera perseguido gustosamente gratis, por pasión. Me siento muy agradecido.

Es difícil imaginar haber vivido un tiempo más emocionante como un devoto de las computadoras. De los mecanismos a los tubos al vacío a los transis-

tores a los circuitos integrados, la tecnología ha explotado. La primera computadora en la cual trabajé, recién salido de Harvard, fue la supercomputadora IBM 7030 Stretch. Ésta reinó como la computadora más rápida de 1961 a 1964; nueve copias fueron liberadas. Mi Macintosh Powerbook es hoy no sólo más rápida, con mayor memoria y disco más grande, es mil veces más barata. (Cinco mil veces más barata en dólares constantes.) Hemos visto a su vez la revolución de las computadoras, la revolución de las computadoras electrónicas, la revolución de las minicomputadoras y la revolución de las microcomputadoras, cada una de ellas aportando más computadoras de órdenes de magnitud.

La disciplina intelectual relacionada con las computadoras ha explotado como lo ha hecho la tecnología. Cuando era un estudiante de posgrado a mediados de los 50's, podía leer *todas* las revista y actas de conferencias; Podía estar al tanto en *todas* las disciplinas. Hoy en día, mi vida intelectual me ha visto con pesar dando el beso de despedida a cada una de las subdisciplinas que me atraen ya que mi carpeta de trabajo se ha desbordada mas allá de su capacidad. Demasiados intereses, demasiadas oportunidades emocionantes por aprender, investigar y pensar. Que lío más maravilloso! No sólo el fin no está a la vista tampoco el ritmo está disminuyendo. Tenemos muchas satisfacciones en el futuro.

Notas y Referencias

Capítulo 1

1. Ershov considera que no es sólo un infortunio, sino también una parte del placer. A. P. Ershov, "Aesthetics and the human factor in programming," *CACM,* 15, 7 (July, 1972), pp. 501-505.

Capítulo 2

1. V. A. Vyssotsky de Bell Telephone Laboratories estima que un gran proyecto puede sostener una acumulación de fuerza de trabajo del 30 por ciento por año. Más allá de eso se tensa e incluso se inhibe la evolución de la estructura informal esencial y sus vías de comunicación discutidas en el Capítulo 7.

   F. J. Corbató del MIT señala que un gran proyecto debe prever un reemplazo de personal del 20 por ciento por año, y estos deben estar técnicamente capacitados e integrados dentro de la estructura formal.

2. C. Portman de International Computers Limited dice. *"Cuando se ha visto que todo funciona y se ha integrado todo, usted tiene cuatro meses más de trabajo por hacer."* Varios otros tipos de divisiones de calendario vienen en Wolverton, R. W., "The cost of developing large-scale software ," *IEEE Trans. on Computers,* C-23, 6 (June, 1974) pp. 615-636.

3. Las figuras de la 2.5 a la 2.8 se deben a Jerry Ogdin, quien al citar mi ejemplo de una publicación anterior de este capítulo mejoró mucho su

ilustración. Ogdin, J. L., "The Mongolian hordes versus superprogrammer," *Infosystem* (Dec., 1972), pp. 20-23

Capítulo 3

1. Sackman, H., W. J. Erikson, and E. E. Grant, "Exploratory experimental studies comparing online and offline programming performance," *CACM,* 11, 1 (Jan., 1968), pp. 3-11.
2. Mills, H., "Chief programmer teams, principles, and procedures," IBM Federal Systems Division Report FSC 71-5108, Gaithersburg, Md., 1971.
3. Baker, F. T., "Chief programmer team management of production programming," *IBM Sys. J.,* 11, 1 (1972).

Capítulo 4

1. Eschapasse, M., *Reims Cathedral,* Caisse Nationale des Monuments Historiques, Paris, 1967.
2. Brooks, F. P., "Arquitectural philosophy," en W. Buchholz (ed.), *Planning A Computer System.* New York: McGraw-Hill, 1962.
3. Blaauw, G. A., "Hardware requirements for the fourth generation" en F. Gruenberger (ed.), *Fourth Generation Computers.* Englewood Cliffs, N.J.: Prentice-Hall, 1970.
4. Brooks, F. P. and K. E. Iverson, *Automatic Data Processing, System/360 Edition.* New York: Wiley, 1969, Capítulo 5.
5. Glegg, G. L., *The Design of Design.* Cambridge: Cambridge Univ. Press, 1969, dice *"A primera vista, la superposición de reglas o principios en la mente creativa parece que estorba más que ayuda, aunque esto es bastante falso en la práctica. El pensamiento disciplinado concentra la inspiración en lugar de cegarla.*
6. Conway, R. W., "The PL/C compiler," *Proceedings of a Conf. on Definition and Implementation of Universal Progamming Languages. Stutttgart, 1970.*
7. Para una buena discusión de la necesidad de una tecnología de la programación, véase C. H. Reynolds, "What's wrong with computer pro-

gramming management?" en G. F. Weinwurm (ed.), *On the Management of Computer Programming.* Philadelphia: Auerbach, 1971, pp. 35-42.

Capítulo 5

1. Strachey, C.,*"Review of Planning a Computer System," Comp. J.,* 5, 2 (July, 1962), pp. 152-153.
2. Esto se aplica sólo a los programas de control. Algunos equipos del compilador en el trabajo de la OS/360 estaban construyendo su tercer o cuarto sistema, y la excelencia de sus productos lo demuestra.
3. Shell, D. L., "The Share 709 system: a cooperative effort"; Greenwald, I. D., y M. Kane, "The Share 709 system: programming and modification"; Boehm, E. M., y T. B. Steel, Jr., "The Share 709 system: machine implementation of symbolic programming"; todo en *JACM,* 6, 2 (April, 1959), pp. 123-140.

Capítulo 6

1. Neustadt, R. E., *Presidential Power.* New York: Wiley, 1960, Capítulo 2.
2. Backus, J. W., "The syntax and semantics of the proposed international algebraic language." *Proc, Intl. Conf. Inf. Proc. UNESCO,* Paris, 1959, publicado por R. Oldenbourg, Munich, y Butterworth, London. Además de esto, una colección completa de artículos sobre el tema está contenido en T. B. Steel, Jr. (ed.), *Formal Language Description Lagnuage for Computer Programming.* Amsterdam: North Holland, (1966).
3. Lucas, P., y K. Walk, "On the formal description of PL/I," *Annual Review in Automatic Programming Language,* New York: Wiley, 1962, Capítulo 2, p. 2.
4. Iverson, K. E., *A Programming Language,* New York: Wiley, 1962, Capítulo 2.
5. Falkoff, A. D., K. E. Iverson, E. H. Sussenguth, "A formal description of System/360," ZBM *Systems Journal,* 3, 3 (1964), pp. 198-261.
6. Bell, C. G., and A. Newell, *Computer Structures.* New York: McGraw-Hill, 1970, pp. 120-136, 517-541.

7. Bell, C. G., comunicación privada.

Capítulo 7

1. Parnas, D. L., "Information distribution aspects of design methodo-logy," Carnegie-Mellon Univ., Dept. of Computer Science Technical Report, February, 1971.
2. Derechos de autor, 1939, 1940 Street & Smith Publications, Derechos de autor de 1950, 1967 por Robert A. Heinlen. Publicado por acuerdo con Spectrum Literary Agency.

Capítulo 8

1. Sackman, H., W. J. Erikson, y E. E. Grant, "Exploratory experimentation studies comparing online and offline programming performance," *CACM*, 11, 1 (Jan., 1968), pp. 3-11.
2. Nanus, B., and L. Farr, "Some cost contributors to large-scale programs," *AFIPS Proc. SJCC,* 25 (Spring, 1964), pp. 239-248.
3. Weinwurm, G. F., "Research in the management of computer programming." Report SP-2059, System Development Corp., Santa Monica, 1965.
4. Morin, L. H., "Estimation of resources for computer programming projects," M. S. thesis, Univ. of North Carolina, Chapel Hill, 1974.
5. Portman, C., comunicación privada.
6. Un estudio no publicado de 1964 por E. F. Bardain muestra que los programadores realizan un 27 por ciento de tiempo productivo. (Citado por D. B. Mayer y A. W. Stalnaker," Selection and evaluation of computer personnel," *Proc. 23rd ACM Conf.*, 1968, p. 661.)
7. Aron, J., comunicación privada.
8. Artículo entregado en una sesión de panel y no está incluido en las *Actas de la AFIPS*
9. Wolverton, R. W., "The cost of developing large-scale software," IEEE *Trans, on computers, C-23,* 6 (June, 1974) pp. 615-636. Este importante

artículo reciente contiene datos acerca de muchos de los temas de este capítulo, además de confirmar las conclusiones de productividad.

10. Corbató, F. J., "Sensitive issues in the design of multi-use systems," conferencia en la apertura de la Honeywell EDP Technology Center, 1968.
11. W. M. Taliaffero también reporta una productividad constante de 2400 declaraciones/año en lenguaje ensamblador, Fortran y Cobol. Véase "Modularity. The key to system growth potential," *Software,* 1, 3 (July 1971) pp. 245-257.
12. E. A. Nelson's System Development Corp. Report TM-3225, *Management Handbook for the Estimation of Computer Programming Costs,* muestra una mejora de la productividad de 3 a 1 para lenguajes de alto nivel (pp. 66-67), aunque sus desviaciones estándar son amplias.

Capítulo 9

1. Brooks, F. P. and K. E. Iverson, *Automatic Data Processing, System/360 Edition.* New York: Wiley, 1969, Capítulo 6.
2. Knuth, D. E. *The Art of Computer Programming,* Vols. 1-3, Reading, Mass.: Addison-Wesley, 1968, ff.

Capítulo 10

1. Conway, M. E., "How do committees invent?" *Datamation,* 14, 4 (April, 1968), pp. 28-31.

Capítulo 11

1. Discurso en Oglethorpe University, el 22 de Mayo de 1932.
2. Un relato esclarecedor de la experiencia de Multics acerca de dos sistemas sucesivos está en F. J. Corbató, J. H. Saltzer, and C. T. Clingen, "Multics – The first seven years," *AFIPS Proc SJCC,* 40 (1972), pp. 571-583.
3. Cosgrove, J., "Needed: a new plannign framework," *Datamation,* 17, 23 (Dec. 1971), pp. 37-39.

4. El asunto del cambio de diseño es complejo, y aquí lo sobresimplifiqué. Véase J. H. Saltzer, "Evolutionary design of complex systems," en D. Eckman (ed.), *Systems: Research and Design.* New York: Wiley, 1961. Sin embargo, cuando todo está dicho y hecho todavía abogo por la construcción de un sistema piloto cuyo descarte está planificado.
5. Campbell, E., "Report to the AEC computer Information Meeting." December, 1970. El fenómeno también es discutido por J. L. Ogdin en "Designing reliable software," *Datamation,* 18, 7 (July, 1972), pp. 71-78. Mis experimentados amigos parecen bastante divididos en partes iguales en cuanto a si la curva finalmente cae nuevamente.
6. Lehman, M., and L. Belady, "Programming system dynamics," viene en ACM SIGOPS Third Symposium on Operating System Principles, October, 1971.
7. Lewis, C. S., *Mere Christianity,* New York: Macmillan, 1960, p. 54.

Capítulo 12

1. Véase también J. W: Pomeroy, "A guide to programming tools and techniques," *IBM Sys, J.,* 11, 3 (1972), pp. 234-254.
2. Landy, B., y R. M. Needham, "Software engineering techniques used in the developmen of the Cambridge Multiple-Access System," *Software,* 1, 2 (April, 1971), pp. 167-173.
3. Corbató, F.J., "PL/I as a tool for system programming," *Datamation,* 15, 5 (May, 1969), pp. 68-76.
4. Hopkins, M., "Problems of PL/I for system programming," IBM Research Report RC 3489, Yorktown Heights, N.Y., August 5, 1971.
5. Corbató, F. J., J. H. Saltzer, y C. T. Clingen, "Multics – the first seven years," *AFIPS Proc SJCC,* 40 (1972), pp. 571-582, *"Sólo una media docena de áreas que fueron escritas en PL/I han sido recodificadas en lenguaje de máquina por motivos de exprimir el máximo rendimiento. Varios programas, originalmente en lenguaje de máquina, han sido recodificados en PL/I para mejorar su mantenimiento."*
6. Mencionar el artículo de Corbató citado en la referencia 3: *"PL/I is here now and the alternatives are still untested."* Aunque vemos una opinión

bastante contraria, bien documentada, en Henricksen, J. O. y R. R. Merwin, "Programming language efficiency in real-time software systems," *AFIPS Proc SJCC,* 40 (1972) pp. 155-161.

7. No todos están de acuerdo. Harlan Mills dice, en una comunicación privada, *"Mi experiencia comienza a decirme que en la producción de la programación la persona que debe poner en la terminal es la secretaria. La idea es hacer de la programación una práctica más pública, bajo el escrutinio común de muchos miembros del equipo, en lugar de un arte privado."*
8. Harr, J., "Programming Experience for the Number 1 Electronic Switching System," el artículo viene en SJCC de 1969.

Capítulo 13

1. Vyssotsky, V. A., "Common sense in designing testable software," conferencia en The Computer Program Test Methods Symposium, Chapel Hill, N.C., 1972. La mayor parte de las conferencias de Vyssotsky están contenidas en Hetzel, W. C. (ed.), *Program Test Methods,* Englewood Cliffs, N. J.: Prentice-Hall, 1972, pp. 41-47.
2. Wirth, N., "Program development by stepwise refinement," CACM 14, 4 (April, 1971), pp. 221-227. Véase también Mills, H. "Top-down programming in large systems," en R. Rustin (ed.). *Debugging Techniques in Large Systems.* Englewood Cliffs, N.J.: Prentice-Hall, 1971, pp. 41-55 and Baker, F. T., "Systems quality through structured programming," *AFIPS Proc FJCC,* 41-I (1972), pp. 339-343.
3. Dahl, O.J., E. W. Dijkstra, and C. A. R. Hoare, *Structured Programming,* London and New York: Academic Press, 1972. Este volumen contiene el tratamiento más completo. Véase también la carta seminal de Dijkstra, "GOTO statement considered harmful," *CACM,* 11, 3 (March, 1968), pp. 147-148.
4. Böhm, C., and A. Jacopini, "Flow diagrams, Turing machines, and languages with only two formation rules," *CACM,* 9, 5 (May, 1966), pp. 366-371.
5. Codd, E. F., E. S. Lowry, E. McDonough, and C. A. Scalzi, "Multiprogramming STRETCH: Feasibility considerations," *CACM,* 2, 11 (Nov., 1959),

pp. 13-17.

6. Strachey, C., "Time sharing in large fast computers," *Proc. Int. Conf. on Info. Processing,* UNESCO (June, 1959), pp. 336-341. Véase también la observación de Codd en la p. 341, donde informó de avances en curso como el propuesto en el artículo de Strachey.
7. Corbató, F. J., M. Merwin-Daggett, R. C. Daley, "An experimental time-sharing system," *AFIPS Proc. SJCC, 2,* (1962), pp. 335-344. Reimpreso en S. Rosen, *Programming Systems and Languages,* New York: Mac-Graw-Hill, 1967, pp. 683-698.
8. Gold, M. M., "A methodology for evaluating time-shared computer system usage," tesis de doctorado, Carnegie-Mellon University, 1967, p. 100.
9. Gruenberger, F., "Program testing and validating," *Datamation,* 14, 7, (July, 1968), pp. 39-47.
10. Ralston, A., *Introduction to Programming and Computer Science,* New York: McGraw-Hill, 1971, pp. 237-244.
11. Brooks, F. P., and K. E. Iverson, *Automatic Data Processing, System/360 Edition,* New York: Wiley, 1969, pp. 296-299.
12. Un buen análisis acerca del desarrollo de especificaciones y de la construcción y prueba del sistema viene en F. M. Trapnell, "A systematic approach to the development of system programs," *AFIPS Proc SJCC,* 34 (1969) pp. 411-418.
13. Un sistema de tiempo real requerirá un simulador de ambiente. Por ejemplo, véase M. G. Ginzberg, "Notes on testing real-time system programs," *IBM Sys. J.,* 4, 1 (1965), pp. 58-72.
14. Lehman, M., and L. Belady, "Programming system dynamics," viene en ACM SIGOPS Third Symposium on Operating System Principles, October, 1971.

Capítulo 14

1. Véase C. H. Reynolds, "What's wrong with computer programming management?" en G. F. Weinwurm (ed.), On *the Management of Computer progamming,* Philadelphia: Auerbach, 1971, pp. 35-42.

2. King, W. R. y T. A. Wilson, "Subjective time estimates in critical path planning – a preliminary analysis," *Mgt. Sci.*, 13, 5 (Jan., 1967), pp. 307-320, y secuela, W. R. King, D. M. Witterrongel, K. D. Hezel, "On the analysis of critical path time estimating behavior," *Mgt. Sci.*, 14, 1 (Sept., 1967), pp. 79-84
3. Para una discusión completa, véase Brooks, F. P., and K. E. Iverson, *Automatic Data Processing, System/360 Edition,* New York: Wiley, 1969, pp. 428-430.
4. Comunicación privada.

Capítulo 15

1. Goldstine, H. H., y J. von Neumann, "Planning and coding problems for an electronic computing instrument," Part II, Vol. 1, informe preparado para la U. S. Army Ordinance Department, 1947; reimpreso en J. von Neumann, *Collected Works,* A. H. Taub (ed), Vol. v., New York. McMillan, pp. 80-151.
2. Comunicación privada, 1957. El argumento está publicado en Iverson, K. E., "The Use of APL in Teaching," Yorktown, N. Y.: IBM Corp., 1969.
3. Otra lista de técnicas para PL/I viene dada por Walter y M. Bohl en "From better to best – tips for good programming," *Software Age, 3,* 11 (Nov., 1969), pp. 46-50.

   Las mismas técnicas pueden ser usadas en Algol e incluso Fortran. D. E. Lang de la Universidad de Colorado tiene un programa en Fortran para formatear llamado STYLE que logra tal resultado. Véase también D. D. McCracken y G. M. Weinberg, "How to write a readable FORTRAN program," *Datamation,* 18, 10 (Oct., 1972), pp. 73-77.

Capítulo 16

1. El ensayo titulado "No Silver Bullet" fue tomado de Information Processing 1986, the Proceedings of the IFIP Tenth World Computing Conference, editado por H. J. Kugler (1986), pp. 1069-76. Reimpreso con el amable permiso de IFIP y Elsevier Science B. V., Amsterdam, Países Bajos.

2. Parnas, D. L., "Designing software for ease of extension and contraction,"IEEE *Trans, on SE, 5,2* (March, 1979), pp. 128-138.
3. Booch, G., "Object-oriented design," in *Software Engineering with Ada.* Menlo Park, Calif.: Benjamin/Cummings, 1983.
4. Mostow, J., ed., Special Issue on Artificial Intelligence and Software Engineering, *IEEE Trans, on SE,* 11, 11 (Nov., 1985).
5. Parnas, D. L., "Software aspects of strategic defense sytems," *Communication of the ACM,* 28, 12 (Dec., 1985), pp. 1326-1335. Tambien en *American Scientist, 73, 5* (Sept.-Oct., 1985), pp. 432-440.
6. Balzer, R., "A 15-year perspective on automatic programming," en Mostow, op. cit.
7. Mostow, op. cit.
8. Parnas, 1985, op. cit.
9. Raeder, G., "A survey of current graphical programming techniques," in R. B. Grafton and T. Ichikawa, eds., Special Issue on Visual Programming, *Computer,* 18, 8 (Aug., 1985), pp. 11-25.
10. Se trata el tema en el Capítulo 15 de este libro.
11. Mills, H. D., "Top-down programming in large systems," *Debugging Techniques in Large Systems,* R. Rustin, ed., Englewood Cliffs, N. J., Prentice-Hall, 1971.
12. Boehm, B. W., "A spiral model of software developmente and enhacement," *Computer,* 20, 5 (May, 1985), pp. 43-57.

Capítulo 17

El material citado sin referencia es de comunicaciones personales.

1. Brooks, F. P., "No silver bullet – essence and accidents of software engineering," in *Information Processing 86,* H, J. Kugler, ed. Amsterdam: Elsevier Science (North Holland), 1986, pp. 1069-1076.
2. Brooks, F. P., "No silver bullet – essence and accidents of software engineering," *Computer* 20, 4 (April, 1987), pp. 10-19.

3. Varias de la cartas, y una respuesta, aparecieron en Julio de 1987 en la revista *Computer*.

   Es especialmente satisfactorio observar que mientras "NEBP" no recibió ningún premio, la revisión del artículo por Bruce M. Skwiersky fue seleccionado como la mejor revisión publicada en *Computing Reviews* en 1988. E. A. Weiss, "Editorial," *Computing Reviews* (June, 1989), pp. 283-284, ambas anuncian el premio y reimprimen el estudio de Skwiersky. La revisión tiene un error significativo: en lugar de "seis veces" debe ser "10^6."

4. "De acuerdo a Aristóteles, y en la filosofía Escolástica, un accidente es una cualidad que no pertenece a la cosa en sí a la esencia o la naturaleza substancial del objeto sino que ocurre dentro de él como un efecto de otras causas." *Webster New International Dictionary of the English Language,* 2d ed., Springfield, Mass.: G. C. Merriam, 1960.

5. Sayers, Dorothy L., *The Mind of the Maker.* New York: Harcourt, Brace, 1941.

6. Glass, R. L., y S. A. Conger, "Research software tasks: Intellectual or clerical?" *Information and Management,* 23, 4 (1992). Los autores reportan una medición de la especificación de requerimientos de software y es aproximadamente 80% intelectual y 20% administrativo. Fjelstadt y Hamlen, en 1979, obtiene esencialmente el mismo resultado para el mantenimiento del software de aplicación. No conozco ningún intento de medir esta fracción para la totalidad de la tarea, de punta a punta.

7. Herzberg, F., B. Mausner, and B. B. Sayderman. *The Motivation to Work,* 2nd ed. London: Wiley, 1959.

8. Cox, B. J., "There is a silver bullet," *Byte* (Oct., 1990), pp. 209-218.

9. Harel, D., "Biting the silver bullet: Toward a brighter future for system development," *Computer* (Jan., 1992), pp. 8-20.

10. Parnas, D. L., "Software aspects of strategic defense systems," *Communications of the ACM,* 28, 12 (Dec., 1985), pp. 1326-1335.

11. Tursky, W. M., "And no philosophers' stone, either," in *Information Processing 86,* H. J. Kugler, ed. Amsterdam: Elsevier Science (North Holland), 1986, pp. 1077-1080.

12. Glass, R. L. y S. A. Conger, "Research Software Tasks: Intellectual or Clerical?" *Information and Management,* 23, 4 (1992), pp. 183-192.
13. *Review of Electronic Digital Computers, Proceedings of a Joint AIEE-IRE Computer Conference* (Philadephia, Dec. 10-12, 1951). New York: American Institute of Electrical Engineers, pp. 13-20.
14. *Ibid.,* pp. 36, 68, 71, 97.
15. *Proceedings of the Eastern Joint Computer Conference,* (Washington, Dec. 8-10, 1953). New York: Institute of Electrical Engineers, pp. 45-47.
16. *Proceedings of the 1955 Western Joint Computer Conference* (Los Angeles, March 1-3, 1955). New York: Institute of Electrical Engineers.
17. Everett, R. R., C. A. Zraket, and H. D. Bennington, "SAGE – A data processing system for air defense," *Proceedings of the Eastern Joint Computer Conference,* (Washington, Dec. 11-13, 1957). New York: Institute of Electrical Engineers.
18. Harel, D. H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A: Shtul-Trauring, "Statemate: A working environment for the development of complex reactive systems," IEEE *Trans, on SE,* 16, 4 (1990), pp. 403-444.
19. Jones, C., *Assesment and Control of Software Risks.* Englewood Cliffs, N. J.: Prentice-Hall, 1994. p. 619.
20. Coqui, H., "Corporate survival. The Software dimension," *Focus '89,* Cannes, 1989.
21. Coggins, James M., "Designing C++ libraries," *C++ Journal,* 1, 1 (June, 1990), pp. 25-32.
22. El futuro es tensión; Aún no conozco de ningún resultado de este tipo que se haya informado para un quinto uso.
23. Jones, op. cit., p. 604.
24. Huang, Weigiao, "Industrializing software production," *Proceedings ACM 1988 Compute Science Conference,* Atlanta, 1988. Temo la falta de crecimiento laboral personal en un arreglo de este tipo.
25. Toda la edición de Septiembre de 1994 de IEEE *Software* está dedicada al reuso.

26. Jones, op. cit., p. 323.

27. Jones, op. cit., p. 329.

28. Yourdon, E., *Decline and Fall of the American Programmer.* Englewood Cliffs, N. J.. Yourdon Press, 1992, p. 221.

29. Glass, R. L., "Glass" (columna), *System Development,* (January, 1988), pp. 4-5.

Capítulo 18

1. Boehm, B. W., *Software Engineering Economics,* Englewood Cliffs, N. J.: Prentice-Hall, 1981, pp. 81-84

2. McCarthy, J., "21 Rules for Delivering Great Software on Time," Software World USA Conference, Washington (Sept., 1994).

Capítulo 19

El material citado sin referencia es de comunicaciones personales.

1. En esta desagradable materia, véase también Niklaus Wirth "A plea for lean Software," *Computer,* 28, 2 (Feb., 1995), pp. 64-68.

2. Coleman, D., 1994, "Word 6.0 packs in features; update slowed by baggage," *MacWeek,* 8, 38 (Sept. 26, 1994), p. 1.

3. Se han publicado muchos estudios acerca del lenguaje de máquina y frecuencias de comando del lenguaje de programación *después* de su puesta en uso. Por ejemplo, véase J. Hennessy and Patterson, *Computer Architecture.* Estos datos de frecuencia son muy útiles para construir productos sucesores, aunque nunca se aplican exactamente. No conozco ninguna publicación con estimaciones de frecuencia preparadas *antes* de que se diseñara el producto, y mucho menos comparaciones de estimaciones *a priori* y datos *a posteriori.* Ken Brooks sugiere que un boletín de anuncios en Internet proporcione ahora un método barato de solicitar datos a los potenciales usuarios de un nuevo producto, aun cuando solo responda un conjunto auto-seleccionado.

4. Conklin, J., y M. Begeman, "gIBIS: A Hypertext Tool for Exploratory Policy Discussion," *ACM Transactions on Office Information Systems,* Oct. 1998, pp. 303-331.
5. Englebart, D., y W. English, "A research center for augmenting human intellect," *AFIPS Conference Proceedings, Fall Joint Computer Conference,* San Francisco (Dec. 9-11, 1968), pp. 395-410.
6. Apple Computer, Inc., *Macintosh Human Interface Guidelines,* Reading, Mass.: Addison-Wesley, 1992.
7. Parece que el Apple Desk Top Bus podía manejar dos ratones electrónicamente, pero el sistema operativo no proporciona dicha funcionalidad.
8. Royce, W. W., 1970. "Managing the development of large software systems: Concepts and techniques," *Proceedings,* WESCON (Aug., 1970), reimpreso en el *ICSE 9 Proceedings.* Ni Royce ni otros creían que se podía pasar a través del proceso de software sin revisar los primeros documentos; el modelo fue presentado como un ideal y una ayuda conceptual. Véase D. L. Parnas y P. C. Clements, "A rational design process: How and why to fake it," IEEE *Transactions on Software Engineering,* SE-12, 2 (Feb., 1986), pp. 251-257.
9. Una importante revisión de DOD-STD-2167 produjo DOD-STD-2167A (1988), que permite aunque no exige modelos más recientes tales como el modelo en espiral. Desafortunadamente, el MILSPECS que 2167A cita y los ejemplos ilustrativos que usa están todavía orientados al modelo en cascada, así que la mayoría de las adquisiciones han continuado usando el modelo en cascada, informa Boehms. A Defense Science Board Task Force bajo Larry Druffel y George Heilmeyer, en su "Report of the DSB task force, on acquiring defense software commercially," ha defendido el uso generalizado de modelos más modernos.
10. Mills, H. D., "Top-down programming in large systems," in *Debugging Techniques in Large Sysems,* R. Rustin, ed. Englewood Cliffs, N.J.: Prentice-Hall, 1971.
11. Parnas, D. L., "On the design and development of program families," IEEE *Trans, on Software Engineering,* SE-2, 1 (March, 1976) pp. 1-9; Parnas,

D. L., "Designing software for ease of extension and contraction," IEEE *Trans, on Software Engineering,* SE-5, 2 (March, 1979), pp. 128-138.

12. D. Harel, "Biting the silver bullet," *Computer* (Jan., 1992), pp. 8-20.
13. Los artículos seminales acerca del ocultamiento de información son: Parnas D. L., "Information distribution aspects of design methodology," Carnegie-Mellon, Dept. of Computer Science, Technical Report (Feb., 1971); Parnas, D. L., "A Technique for software module specification with examples," *Comm. ACM,* 5, 5 (May, 1972), pp. 330-336; Parnas, D. L. (1972). "On the criteria to be used in decomposing systems into modules," *Comm. ACM,* 5, 12 (Dec., 1972), pp. 1053-1058.
14. Las ideas de objetos fueron inicialmente esbozadas por Hoare y Dijkstra, pero el primer y más influyente desarrollo de ellas fue el lenguaje Simula-67 de Dahl y Nygaard.
15. Boehm, B. W., *Software Engineering Economics,* Englewood Cliffs, N.J.: Prentice-Hall, 1981, pp. 83-94; 470-472.
16. Abdel-Hamid, T., and S. Madnick, *Software Project Dynamics: An Integrated Approach,* ch. 19, "Model enhacement and Brooks's law." Englewood Cliffs, N.J.: Prentice-Hall, 1991.
17. Stutzke, R. D., "A Mathematical Expression of Brooks's Law." In *Ninth International Forum on COCOMO and Cost Modeling.* Los Angeles: 1994.
18. DeMarco, T., and T. Lister, *Peopleware: Productive Projects and Teams.* New York: Dorset House, 1987.
19. Pío 11, Encyclical *Quadragesimo Anno,* [Ihm, Claudia Carlen, ed., *The Papal Encyclicals 1903-1939,* Raleigh, N.C.: McGrath, p. 428.]
20. Schumaher, E. E., *Small is Beautiful: Economics as if People Mattered,* Perennial Library Edition. New York: Harper and Row, 1973, p. 244.
21. Schumacher, op. cit., p. 34.
22. Un póster que llama a pensar proclama: "La libertad de prensa pertence a aquel que tiene una"
23. Bush, V., "That we may think," *Atlantic Montly,* 176, 1 (April, 1945), pp. 101-108.

24. Ken Thompson de Bells Labs, inventor de Unix, se dió cuenta pronto de la importancia de la pantalla grande para la programación. Ideó una manera de obtener 120 línea de código, en dos columnas, en su primitivo tubo de almacenamiento electrónico Tektronix. Se aferró a esta terminal a lo largo de toda una generación de pequeñas ventanas y tubos veloces.

Índice