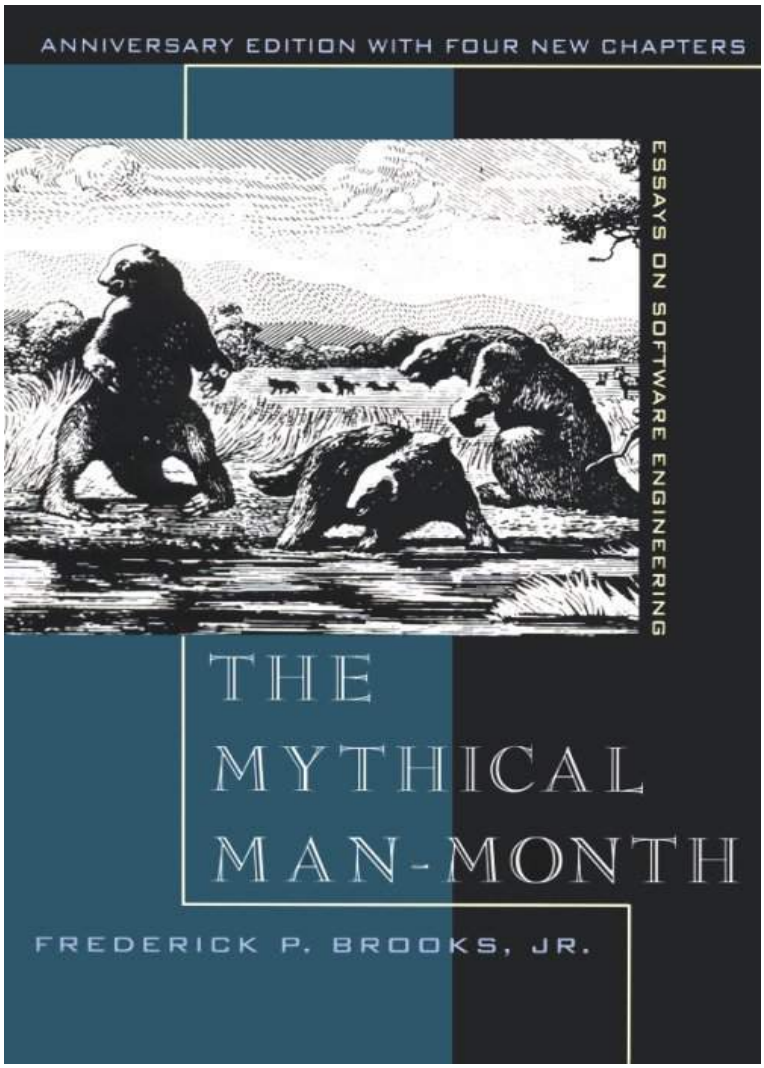

ANNIVERSARY EDITION WITH FOUR NEW CHAPTERS
ESSAYS ON SOFTWARE ENGINEERING
THE MYTHICAL MAN-MONTH
FREDERICK P. BROOKS, JR.

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 director de proyecto para su desarrollo y más tarde como director del proyecto de software del Operative System/360 en su fase de diseño. 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
anunez20@gmail.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: Esencia y Accidentes de la Ingeniería de Software", un artículo publicado en 1986 para la conferencia 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 progra-

mación 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 ha impresionado cuán pocas afirmaciones del libro han sido criticadas, demostradas, o refutadas por la actual investigación y experiencia de la ingeniería de software. Esto me resulta útil ahora 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 produc-

ción técnica del artículo imprimido aquí como el Capítulo 16. El análisis de los problemas del software en las 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 sólo 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 sólo 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. Corbato 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 escrito como ensayos separables, existe un tema princi-

pal 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 dispersa. 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.

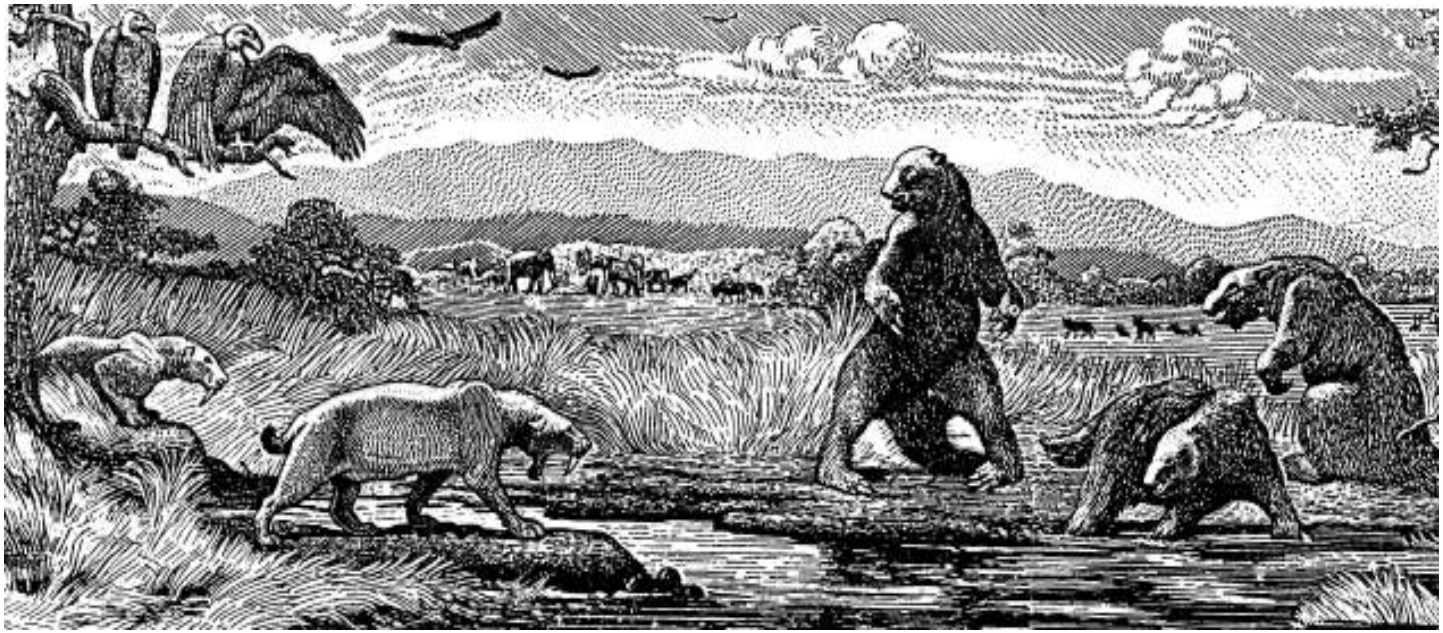
Debido a que este es un libro de ensayos y no uno de texto, todas las referencias y notas han sido desterradas al final del libro y se exhorta al lector a pasarlas por alto en una primera lectura.

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

Ninguna escena de la prehistoria es 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 ésta, y ninguna bestia es tan fuerte o tan hábil para que finalmente no se hunda.

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 para solucionar el problema debemos tratar de entenderlo.

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

El Producto de los Sistemas de Programación

De vez en cuando leemos relatos en las noticias 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 *éso* que se está produciendo.

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

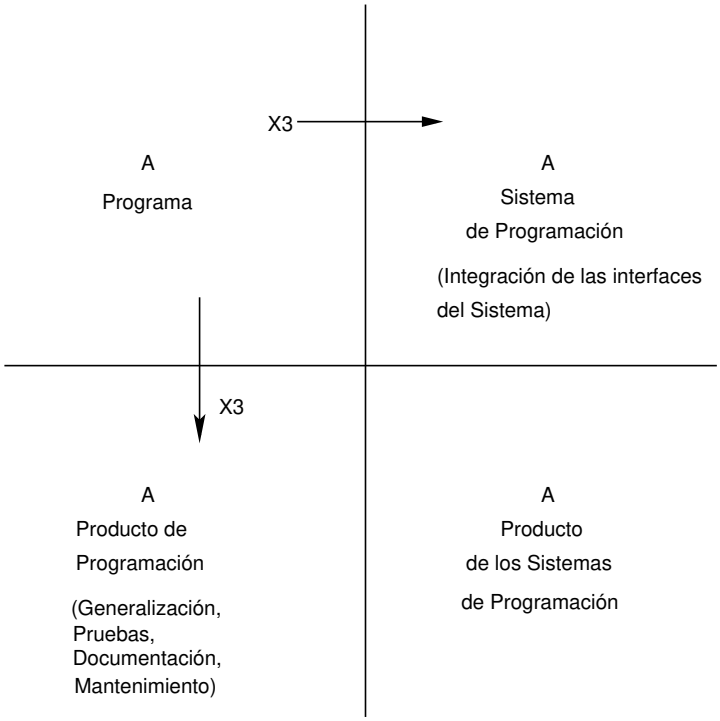


Fig. 1.1 Evolución del producto de los sistemas de programación

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. Es utilizable 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 contar con 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. De modo empírico, calculo 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, coordinados en funcionamiento y disciplinados en formato, tal que todo el ensamblaje constituya un instrumento completo para realizar 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 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, debido al número de casos que crece de forma combinatoria. Todo esto consume tiempo, debido a los sutiles errores que surgen de las interacciones inesperadas de los 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 de los programas simples en todas las formas de arriba. Cuesta nueve veces más. Pero realmente es un objeto útil, el producto planeado por el trabajo de gran parte de la programación de sistemas.

Las Satisfacciones del Oficio

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

Primero, la pura satisfacción de hacer cosas. Así como los niños se deleitan haciendo sus pasteles de lodo, así los adultos se divierten construyendo cosas, especialmente cosas de sus propios diseños. Pienso que esta satisfacción debe ser una imagen de la satisfacción de dios de construir cosas, una satisfacción demostrada en la individualidad y originalidad de cada hoja y cada copo de nieve.

Segundo, es la satisfacción de hacer cosas que sean útiles para otras personas. En lo más profundo, queremos que otros usen nuestro trabajo y lo encuentren útil. En este sentido la programación de sistemas no es esencialmente diferente al primer recipiente de arcilla del niño para los lápices "de la oficina de papá".

Tercero, es la fascinación de crear objetos complejos parecidos a rompecabezas, de entrelazar piezas móviles y observarlos trabajar en ciclos ingeniosos, ejecutando desde el inicio los resultados de sus fundamentos incorporados. Una computadora programada tiene toda la fascinación del juego de pinball o del mecanismo de la máquina de discos, llevada al extremo.

Cuarto, es la satisfacción del continuo aprendizaje, el cual 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 del pensamiento puro. 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 reutilizar, tan fácilmente capaces de llevar a cabo estupendas estructuras conceptuales. (Como veremos después, esta gran manipulabilidad 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 trabaja, y produce salidas visibles separadas de la construcción misma. Imprime resulta-

dos, traza dibujos, produce sonidos, mueve brazos. La magia del mito y la leyenda se hace realidad en nuestro tiempo. Uno teclea el conjuro correcto, y la pantalla cobra vida, mostrando cosas que jamás existieron ni podrían existir.

Por lo tanto, la programación es divertida porque gratifica el anhelo creativo inmerso en lo profundo de nosotros y alegra 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 ayudarán a 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 de aprender a programar.[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 para toda nuestra responsabilidad. Sin embargo, parece que en todos los campos, donde se lleva a cabo el trabajo productivo no se tiene una autoridad formal proporcional 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 desagradable para el programador de sistemas. Pues éste depende de los programas de otras personas. Y estos están generalmente 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 ser utilizables.

El siguiente infortunio es que diseñar conceptos estupendos es divertido; encontrar una plaga de pequeños errores es sólo trabajo.

Acompañado de cualquier trabajo creativo vienen pesadas horas de tedio, de meticuloso trabajo, 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. Así, las pruebas se hacen interminables, encontrar los últimos errores difíciles toma más tiempo que encontrar los primeros.

Y para empeorar las cosas, el último infortunio es que el producto en el que se ha trabajado durante tanto tiempo parece ser obsoleto al finalizar (o antes). Los colegas y competidores ya están en la intensa búsqueda de nuevas y mejores ideas. La actividad del pensamiento infantil ya no solo está concebida sino calendarizada.

Esto siempre parece peor de lo que en realidad 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 trabaja avanza *siempre*. Tan pronto como se congela el diseño, llega a ser obsoleto en términos de sus conceptos. Pero la implementación de productos reales exige coordinación 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 los infortunios, y para ellos el resto de este libro intentará tender algunos puentes 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 berre .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
Liedetkranz .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 carencia de un 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 irá bien.

Segundo, nuestras técnicas de estimación confunden de forma equivocada el esfuerzo con el progreso, y ocultan esa suposición de que los hombres y los 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 disciplinas de la ingeniería.

Quinto, cuando se detecta una demora 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 es 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. Pero 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 irá 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 el trabajo creativo 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 "el desarrollo" son prácticas esenciales para el teórico.

En muchos trabajos creativos el medio de ejecución es inmanejable. Maderas partidas; manchas de pintura; cortos en circuitos eléctricos. Estas limitaciones físicas del medio restringen las ideas que se pueden expresar, y además crean dificultades inesperadas a la hora de su 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 al medio físico por la mayoría de nuestras dificultades de implementación; porque los medios no son "nuestros" en la forma en que nuestras ideas lo son, 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 del pensamiento puro: los conceptos y sus tan flexibles 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 incorrectas, cometemos errores; por lo tanto, nuestro optimismo está injustificado.

Dada una tarea, la suposición de que todo irá bien tiene un efecto

probabilístico en el calendario. Y efectivamente la tarea podría ir como está planeada, pues encontraremos que existe una distribución de probabilidad para el retraso, y asimismo que "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 vaya bien será cada vez más pequeña.

El Hombre-Mes

El segundo modo del pensamiento falaz se expresa en la propia unidad de esfuerzo utilizada en la estimación y 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 la unidad del hombre-mes como 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 mercancías 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.

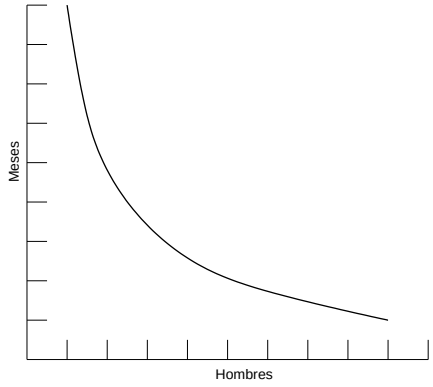


Fig. 2.1 Tiempo versus número de trabajadores – una tarea perfectamente divisible

Cuando una tarea no puede dividirse debido a las 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 del depurado.

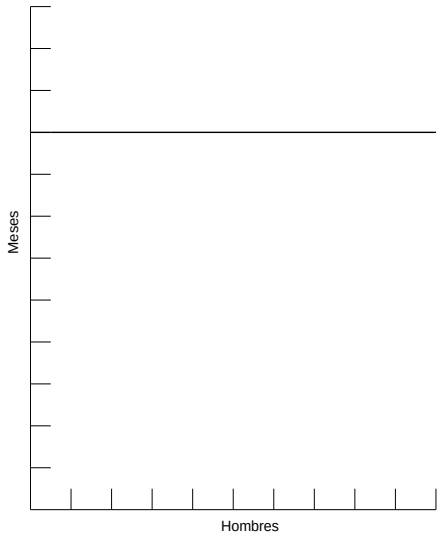


Fig. 2.2 Tiempo versus número de trabajadores – una tarea no divisible

En tareas que pueden dividirse pero que requieren comunicación entre 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 aún que el intercambio entre hombres y meses (Fig. 2.3).

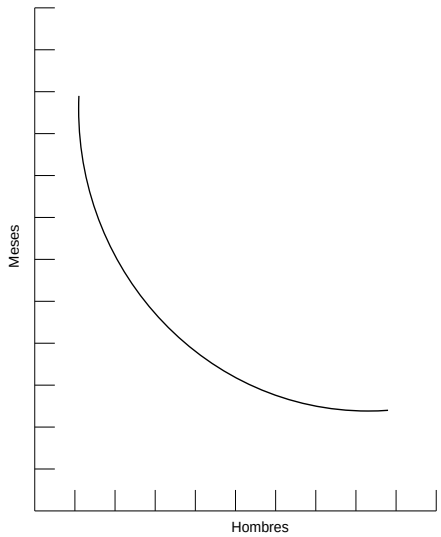


Fig. 2.3 Tiempo versus número de trabajadores – una tarea divisible que requiere comunicación

La carga adicional de comunicación está compuesta de dos partes, entrenamiento e intercomunicación. Cada trabajador debe ser entrenado en la tecnología, los objetivos del proyecto, la estrategia global y el plan de trabajo. Este entrenamiento no se puede dividir, así que esta parte del esfuerzo añadido varía linealmente con el número de trabajadores.[1]

La intercomunicación es peor. Si cada parte de la tarea debe coordinarse de forma separada con cada una de las otras partes el esfuerzo se incrementa como $n(n-1)/2$. Tres trabajadores requieren tres veces más pares de intercomunicación que dos; cuatro trabajadores requieren 6 veces más que dos. Si además, se necesitan reuniones de trabajo entre tres, cuatro, etc., trabajadores para resolver cosas juntos, el asunto todavía se pone peor. 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 trabajo de sistemas – una operación de interrelaciones complejas – la labor de comunicación es grande, y domina rápidamente la reducción del tiempo de cada tarea, obtenida a través de la división. Por lo tanto, añadir más personas no reduce sino extiende el calendario.

Pruebas del Sistema

No hay parte del calendario más afectada por las restricciones secuenciales que la depuracion de componentes y la prueba del sistema. Además, el tiempo requerido depende de la cantidad y lo escurridizo de los errores. Teóricamente este número debería ser cero. En virtud de nuestro optimismo, generalmente esperamos que el número de errores sea menor de lo que finalmente resultan 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
1/4 pruebas del sistema con todos los componentes disponibles

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

1. La fracción dedicada a la planificación es mayor a la 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 está dedicada al depurado de programas terminados y es mucho mayor a lo normal.
3. A la parte que es fácil de calcular, 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 estos cumplían con el calendario hasta y exceptuando en las pruebas del sistema.[2]

En concreto, errar en asignar tiempo suficiente a pruebas del sistema es especialmente desastroso. Puesto que los retrasos vienen al final del calendario, nadie se da cuenta de los problemas hasta casi la fecha de entrega. Malas noticias, retrasado y sin aviso previo, eso 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 por el retraso son muy altos, pues casi es hora del envío del software. Y efectivamente, estos costos secundarios pueden sobrepasar por mucho a los demás. Por lo tanto, es muy importante disponer de tiempo suficiente para pruebas del sistema en el

calendario original.

Estimación sin Agallas

Note que para el programador, como para el cocinero, la urgencia del cliente puede guiar la conclusión de la tarea, pero no puede guiar su conclusión real. Una tortilla, prometida en dos minutos, puede parecer una muy buen mejora. Pero cuando no ha cuajado en dos minutos, el cliente tiene dos opciones – esperar o comerlo crudo. Los clientes del software tienen las mismas opciones.

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

Ahora bien, no creo que los gestores de software tengan menos coraje propio y firmeza que los cocineros, ni que otros gestores de la ingeniería. Pero los falsos calendarios con tal de cumplir con la fecha que desea el cliente 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 ocurrencias de los gestores.

Claramente necesitamos dos soluciones. Tenemos que desarrollar y difundir estadísticas de productividad, estadísticas 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 imprescindible está retrasado? Añadir fuerza laboral, 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 suponga que no se cumplió la primera etapa hasta que transcurrieron dos meses (Fig. 2.6). ¿Cuáles son las alternativas que encara el gestor?

1. Suponga 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 esfuerzo, y dos meses, así que se necesitarán $4\frac{1}{2}$ personas. Hay que añadir 2 personas a las 3 ya asignadas.

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

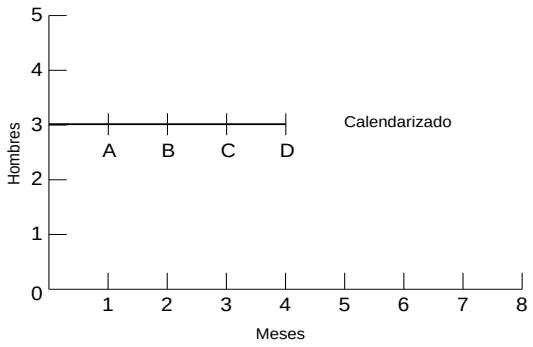


Figura 2.5

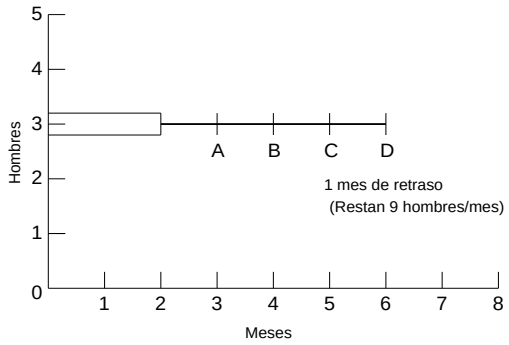


Figura 2.6

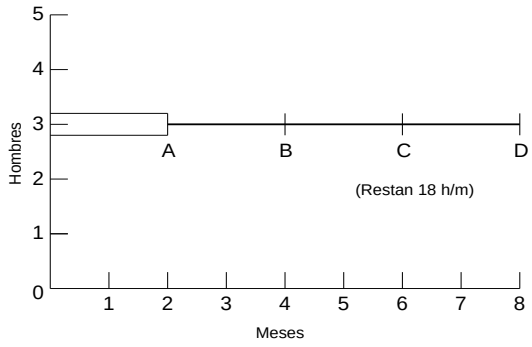


Figura 2.7

3. Recalendarice. Me gusta el consejo dado por P. Fagg, un experimentado ingeniero de hardware, "No cometa errores pequeños." Eso significa, permitir tiempo suficiente en el nuevo calendario para asegurar que el trabajo se lleve a cabo completa y cuidadosamente, y no se tenga que recalendarizar de nuevo.

4. Recorte la tarea. En la práctica esto tiende a suceder de cualquier manera, una vez que el equipo observa la demora 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 primeros dos 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, que a pesar de ser competentes y haber sido reclutadas a prisa, requerirán ser entrenadas en la tarea por una persona con experiencia. Si esto toma un mes, *3 hombres-mes habrán sido dedicados 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 entrenadas y un mes. Como la Fig. 2.8 sugiere, el producto está retrasado como si nadie hubiese sido añadido (Fig. 2.6).

El deseo de llevarlo a cabo en cuatro meses, considerando solo el tiempo de entrenamiento sin ninguna repartición ni pruebas extras de los sistemas, requeriría añadir al final del segundo mes, 4 personas, no 2. Para lograr la repartición y el efecto de las pruebas del sistema, se tendría que añadir otra persona más. Sin embargo, ahora al menos se tiene 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.

Note que por el final de tercer mes el panorama es sombrío. El hito

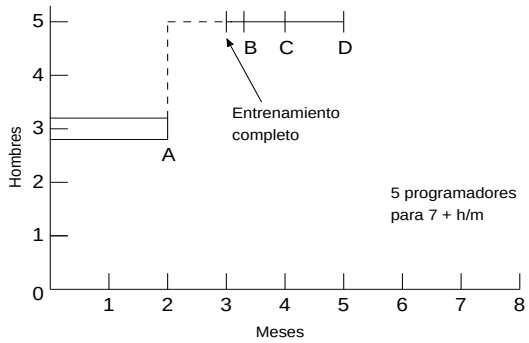


Figura 2.8

del 1 de Marzo no ha sido alcanzado a pesar de todo el trabajo de gestión. La tentación de repetir el ciclo es muy fuerte, añadir aún más mano de obra. Ahí yace la locura.

Lo anterior suponía que solo el primer hito fue mal estimado. Si en Marzo se hace la suposición conservadora de que todo el calendario fue optimista, como muestra la Fig . 2.7, se busca añadir 6 personas solo a la tarea original. El cálculo de los efectos del entrenamiento, de la repartición y las pruebas del sistema se dejan al lector como ejercicio. Sin duda alguna, el desastre regenerativo conducirá a un producto más pobre, más retrasado, que si se hubiera recalendarizado la tarea 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 puede obtener un calendario capaz de funcionar usando más personas y menos meses. Gran parte de los proyectos de software han fracasado más por la carencia de un 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

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. Sólo dentro de este grupo, la razón promedio del rendimiento entre los mejores y los peores programadores era cerca de 10:1 en las mediciones de productividad y un impresionante 5:1 en las mediciones de la 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 podría 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 de las mentes afecta el costo del trabajo, ya que una gran parte de éste es el costo de la comunicación y la corrección de los efectos adversos de la mala comunicación (depurado del sistema). Esto, también, sugiere que se busque 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 experimenta-

dos, 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. Por ejemplo, considere el OS/360. En el punto más alto trabajaban más de 1000 personas en él – programadores, escritores, operadores, empleados, 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 pequeño para sistemas realmente grandes*. Considere el trabajo del OS/360 como si pudiera 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 a 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. Para la 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 poder 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. Esto es, en lugar de que cada miembro haga cortes, sólo 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 la desiderata, 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? ¿Quién será anestesiólogo? 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 sólo 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 un pensador, un debatidor y un evaluador. El cirujano le sugiere ideas, aunque no está sujeto a sus consejos. Normalmente, el copiloto es el representante de su equipo en

discusiones con otros equipos acerca del funcionamiento 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 el responsable por ninguna parte del mismo.

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

El editor. El cirujano es el responsable de generar la documentación – para una máxima claridad es necesario que él la escriba. Esto se aplica tanto para las descripciones internas y como para las externas. Sin embargo, el editor toma el borrador o el manuscrito elaborado por el cirujano y lo analiza, lo reelabora, le añade referencias y bibliografía, lo mantiene a través de sus distintas versiones, y supervisa su mecanismo 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 del mantenimiento de todos los registros técnicos del equipo dentro de una biblioteca de productos de programación. El archivista está entrenado como una secretaria y es el responsable de los archivos legibles tanto por la computadora como por los humanos.

Todo ingreso a la computadora va al archivista, quien lo registra y tipea en caso necesario. Asimismo, obtiene los listados de salida para ser archivados e indexados. Las ejecuciones más recientes de cualquier modelo se mantienen en una libreta 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 hacer claras *todas* las operaciones de la computadora a todos los miembros del equipo y señala que todos los programas y datos son propiedad del equipo, y no propiedad privada.

La labor especializada del archivista releva al programador de las tareas de oficina, sistematiza y asegura un desempeño adecuado en esas tareas que con frecuencia se eluden, y mejora el activo más valioso del equipo – el producto del trabajo. Claramente el concepto expuesto anteriormente asume ejecuciones por lotes. Cuando se usan terminales interactivas, particularmente aquellas con salidas no impresas, las funciones del archivista no disminuyen, sino que cambian. Ahora registra todas las actualizaciones de las copias de los programas 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 depurado interactivo, de tal manera que un equipo rara vez necesitará su propia máquina y personal de operadores. Estos servicios deben estar disponibles con una respuesta y una fiabilidad indiscutiblemente satisfactorias; 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 la adecuación del servicio básico y de la construcción, mantenimiento, y actualización de herramientas especiales que el equipo necesita – en su mayoría serán servicios de cómputo interactivo. 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 utilerías especializadas, procedimientos catalogados, 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. Por lo tanto, el probador es 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 el depurado diario. También podría planificar secuencias de pruebas y construir el andamiaje necesario para 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 con 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 servir 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 organizado de esta manera cumple con la desiderata 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 discos, etc. Y también asegura la integridad conceptual del trabajo.

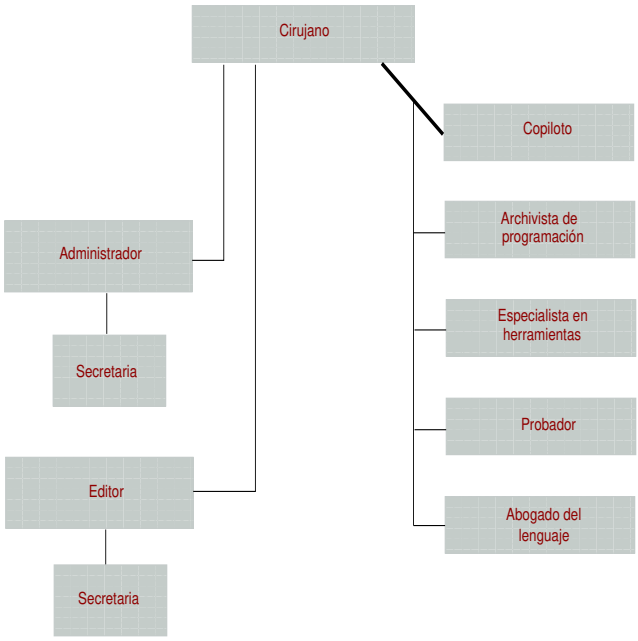


Fig. 3.1 Patrón de comunicación en equipos de programación de 10 personas

Segundo, en el equipo convencional los participantes son iguales, las inevitables diferencias de 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 a la interfaz, aunque estén compuestas por diferencias de interés – e.g., a quienes 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 carencia de división del problema y la relación superior-subordinado hacen posible que el equipo quirúrgico actúe *uno animo*.

Incluso la especialización de funciones del resto del equipo es la clave para su eficiencia, y permite un patrón de comunicación radical-

mente 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.

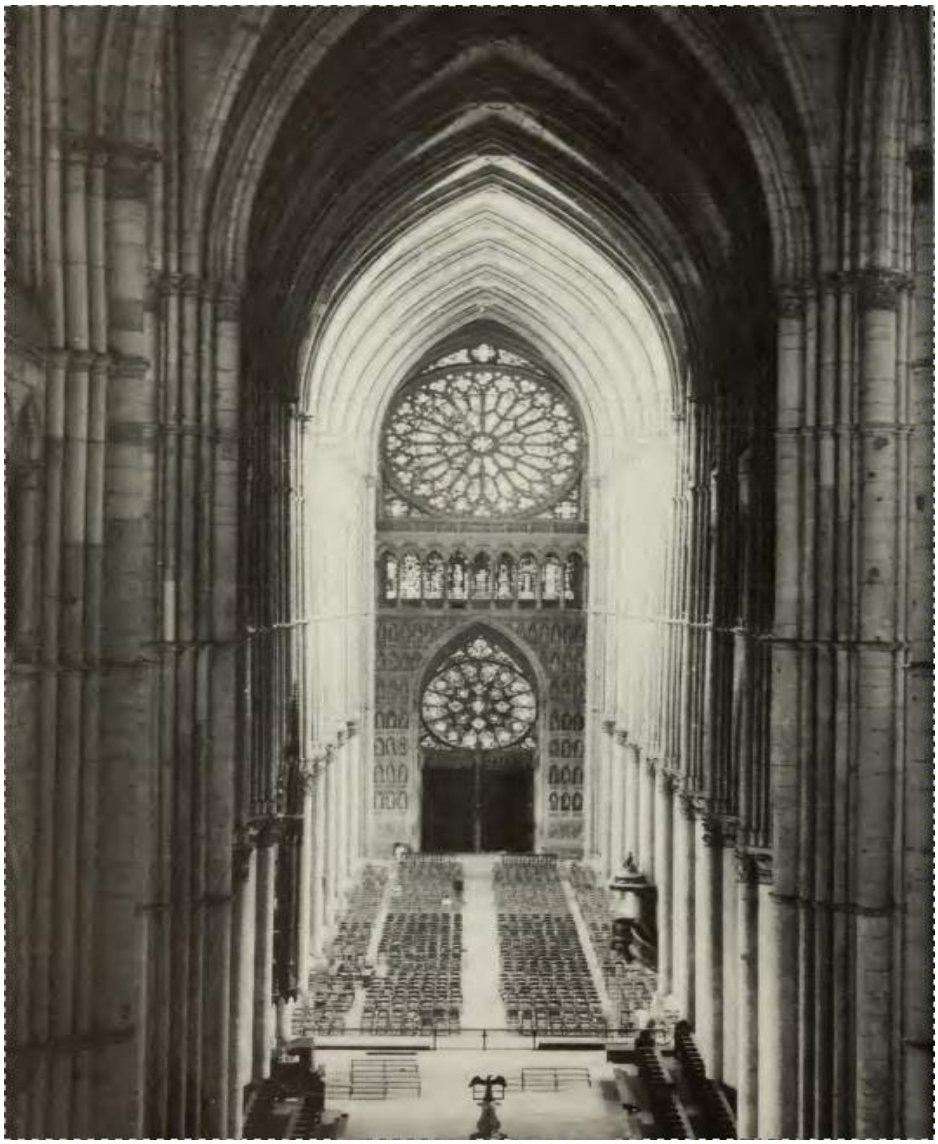
A Mayor Escala

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 a mayor escala 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 sólo 20 mentes, las de los cirujanos.

Sin embargo, para ese problema de coordinación se deben usar técnicas distintas, que discutimos en los siguientes capítulos. Por el momento 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. De cualquier manera, tales roles y técnicas han mostrado 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.
Este 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 en sus diversas partes en el plan o en el estilo arquitectónico, pues fueron realizadas por diferentes constructores y en distintas épocas. Los últimos constructores fueron tentados para "mejorar" el diseño 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.

Contra 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, 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 dispongamos 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 pero 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?
- ¿Este argumento acaso no implica tener una élite, o una aristocracia de arquitectos, y una horda de implementadores plebeyos

cuyos talentos creativos e ideas han sido suprimidos?

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

Lograr la Integridad Conceptual

El propósito de los sistemas de programación es construir computadoras que sean fáciles 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 programas 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. Así, el usuario encontrará que es mucho más fácil especificar cualquier función en particular, 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 funcional excede el tiempo perdido en aprender, recordar, y buscar manuales. Con los sistemas de programación modernos esta ganancia supera el costo, aunque en los años recientes la proporción de ganancia a costo parece haber disminuido conforme se fueron agregando funciones más complejas. Estoy impresionado por la facilidad de uso de la memoria de la IBM 650, incluso sin un programa ensamblador o ningún otro tipo de software.

Como el propósito es la facilidad de uso, 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 fue siempre 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, por cualquier medición su funcionamiento no está 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 alcanzan a cumplir 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 instrumentos 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 la misma filosofía y el mismo equilibrio de desiderata. 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 fuerzan a que la construcción del sistema requiera mucha manos. Se dispone 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 separar la arquitectura de la implementación. Yo mismo lo he visto aplicado 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 no aplicarlo 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 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 toda su labor.

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 ha aprendido esta arquitectura, puede decir la hora tan fácilmente desde un reloj de pulsera hasta una torre de iglesia. Sin embargo, la implementación y su realización, describen qué ocurre dentro del empaque – impulsado y controlado con exactitud por una variedad de mecanismos.

Por ejemplo, en el System/360 una sola arquitectura de computadora se implementó de forma muy diferente en cada uno de los nueve modelos. Al contrario, una sola implementación del flujo de datos, la memoria, y el microcódigo de la Model 30, se usaron 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 Fortran IV estándar. 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 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 es tener integridad conceptual, alguien debe controlar los conceptos. Esa es una aristocracia que no necesita justificación.

No, porque el establecer especificaciones externas no es un trabajo más creativo que el diseño de las 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 tanto brillantez técnica como 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, "Forma es liberación." Los peores edificios son aquellos cuyos presupuestos eran demasiado grandes 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 hubiera tenido una mejor arquitectura si hubiera estado más restringida. Las restricciones impuestas por el presupuesto a 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. Es más, éste 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 lo 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 Hacen los Implementadores?

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

El director 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 director 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 director 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 peor calidad. Lo hice, y así fue. Estaba en lo cierto en ambas cuentas. 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 la decisión errada; pero el más abrumador fue el calendario y lo atractivo de poner a esos 150 implementadores a trabajar. Es este canto de sirenas de cuyo riesgo fatal ahora soy conciente.

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 funciones y no reflejarían las consideraciones prácticas del costo.
- Los arquitectos obtendrían toda la diversión creativa y excluirían la inventiva de los implementadores.
- La mayoría de los implementadores tendrían que sentarse ociosamente a que llegaran las especificaciones a través del estrecho embudo que era 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 puras fantasías. 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 sincronización y coordinación. 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 se pueden traslapar 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 diseñando los flujos de datos, las secuencias de control, los conceptos de la densidad del empaque, etc. Idea o adapta las herramientas necesarias, especialmente el sistema de registro, incluyendo el sistema de automatización del diseño.

Mientras tanto, al nivel de la realización, hay que 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.

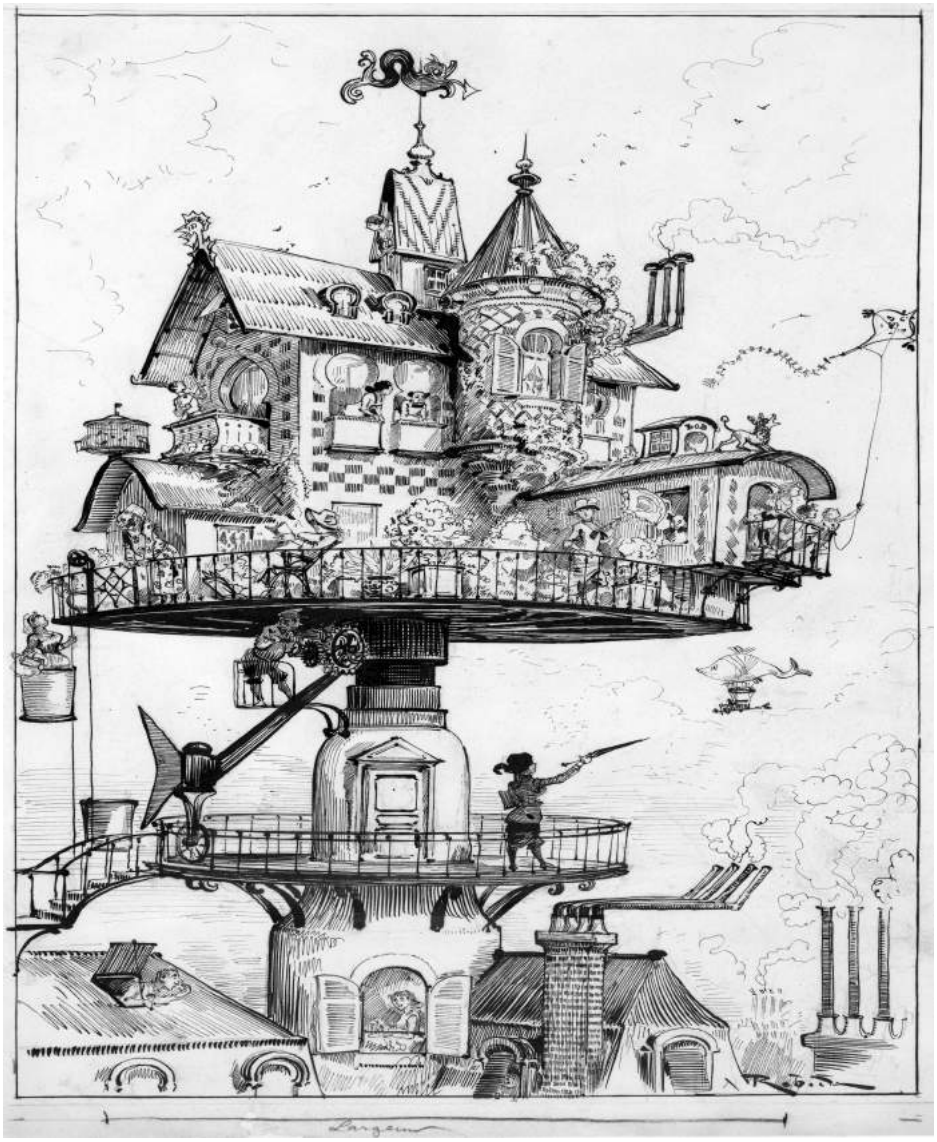
Lo mismo es cierto en el diseño del sistema de programación. Mucho antes de que se completen las especificaciones externas, el implementador tiene mucho trabajo. Puede proceder, dadas algunas vagas aproximaciones en cuanto a las funciones del sistema que finalmente se incorporarán en las especificaciones externas. Debe tener bien definidos los objetivos de espacio y tiempo. Debe conocer la configuración del sistema en el que se ejecutará su producto. Después puede empezar diseñando el alcance de los módulos, las estructuras de tablas, los desgloses de paso o fase, los algoritmos, y todo tipo de herramientas. También, debe invertir algo de tiempo en la comuni-

cación con el arquitecto.

Mientras tanto, en el nivel de la realización también hay mucho que 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 las especificaciones funcionales de la responsabilidad de construir un producto rápido y barato, ¿Qué mecanismo limita el entusiasmo inventivo del arquitecto?

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

Procedimiento Interactivo para el Arquitecto

El arquitecto de una construcción trabaja en contra de un presupuesto, usa técnicas de estimación que luego serán 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. Él, sin embargo, tiene 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 impone;
- 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 las sugerencias de cambiar la arquitectura. A menudo estará en lo cierto – alguna 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 cuidadosamente y con grandes restricciones.

En tanto diseña el primer trabajo, se le ocurren adorno tras adorno y embellecimiento tras embellecimiento. Todos esto queda almacenado para ser usado la "siguiente vez". Tarde o temprano el primer sistema está terminado, y el arquitecto, con firmeza, seguridad y una experiencia demostrada en tal clase de sistemas, está listo para construir el 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 en cuanto a las características generales de tales sistemas, y sus diferencias identificarán aquellas partes de su experiencia que son particulares y no generalizables.

La tendencia general es sobrediseñar el segundo sistema, utilizando toda 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, después incorporada en la 7090. Ésta es una actualización, un segundo sistema de la muy exitosa y limpia 704. El conjunto de operaciones es tan rico y profuso que sólo cerca de la mitad se usaba regularmente.

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

deseos 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 el Stretch es de alguna manera el fin de una tendencia de desarrollo. Como algunos primeros programas de computación es inmensamente ingenioso, inmensamente complicado, y extremadamente eficaz, pero de alguna manera al mismo tiempo crudo, excesivo 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 cambio de fecha para el apropiado manejo del 31 de Diciembre en los años bisiestos (cuando es el Día 366). Esto pudo haberse dejado al operador.

El efecto del segundo-sistema tiene otra manifestación algo distinta del adorno funcional puro. 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 diseñarlo en el código fuente. Permite cambiar la estructura de la superposición de ejecución a ejecución sin recompilación. Facilita una rica variedad de opciones útiles y otros medios. 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 es el supuesto principal. Esto entra en un conflicto directo con la noción de usar superposiciones estáticas. Cuánto mejor funcionaría el sistema si el esfuerzo dedicado al manejo de las 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 por sí mismo muchas superposiciones que aun cuando se usa sólo para enlazar sin aplicar 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. Es como el patinador cuyo estómago se adelante a sus pies, la mejora avanzó hasta que las suposiciones acerca del sistema fueron totalmente rebasadas.

El depurador 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 de volcado de memoria. Usa el concepto de sección de control y una ingeniosa técnica generadora que permite el trazado selectivo y la copia del estado del sistema sin la sobrecarga del intérprete o la 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 depuracion 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 principalmente 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 obviadas 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 asegurarse que los conceptos filosóficos y los objetivos se reflejen totalmente en los detalles del diseño.