

Crédito da foto: © Jerry Markatos

CERCA DE O AUTOR

Frederick P. Brooks, Jr., é Kenan Professor de Ciência da Computação na University of North Carolina em Chapel Hill. Ele é mais conhecido como o "pai do IBM System / 360", tendo atuado como gerente de projeto para seu desenvolvimento e posteriormente como gerente do projeto de software Operating System / 360 durante sua fase de design. Por esse trabalho, ele, Bob Evans e Erich Bloch receberam a Medalha Nacional de Tecnologia em 1985. Anteriormente, ele foi arquiteto dos computadores IBM Stretch e Harvest.

Em Chapel Hill, o Dr. Brooks fundou o Departamento de Ciência da Computação e o presidiu de 1964 a 1984. Ele atuou no National Science Board e no Defense Science Board. Seu ensino e pesquisa atuais são em arquitetura de computadores, gráficos moleculares e ambientes virtuais.

The Mythical Man-Month
Essays on Software Engineering
Anniversary Edition

Frederick P. Brooks, Jr.

Universidade da Carolina do Norte em Chapel Hill



ADDISON-WESLEY

Boston • São Francisco * Nova York «Toronto» Montreal
Londres «Munique * Paris e Madrid
Cidade do Cabo • Sydney • Tóquio • Cingapura • Cidade do México

Desenho da capa: CR Knight, Mural dos Poços de Alcatrão de La Brea.
Cortesia do Museu George C. Page de La Brea Discoveries, Museu de História Natural do Condado de Los Angeles. Designer da capa: Diana Coe.

O ensaio intitulado, No Silver Bullet, é do processamento de informações 1986, Proceedings of the IFIP TenthWorld Computing Conference, editado por H.-J. Kugler, 1986, páginas 1069-1076. Reproduzido com a gentil permissão de IFIP e Elsevier Science BV, Amsterdã, Holanda.

Dados de Catalogação na Publicação da Biblioteca do Congresso

Brooks, Frederick P., Jr. (Frederick Phillips)

O mítico homem-mês: ensaios sobre engenharia de software /

Frederick P. Brooks, Jr. - Edição de aniversário.

p. cm.

Inclui referências bibliográficas e índice.

ISBN0-201-83595-9

1. Engenharia de software. I. Título.

QA76.758.B75 1995

005,1'068 - dc20

94-36653

CIP

Muitas das designações usadas por fabricantes e vendedores para distinguir seus produtos são reivindicadas como marcas registradas. Onde essas designações aparecem neste livro, e Addison-Wesley estava ciente de uma reivindicação de marca registrada, as designações foram impressas em maiúsculas iniciais ou todas em maiúsculas.

Copyright © 1995 Addison Wesley Longman, Inc.

Todos os direitos reservados. Nenhuma parte desta publicação pode ser reproduzida, armazenada em um sistema de recuperação ou transmitida de qualquer forma ou por qualquer meio, eletrônico, mecânico, fotocópia, gravação ou outro, sem a permissão prévia por escrito do editor e do autor. Impresso nos Estados Unidos da América.

Texto impresso em papel reciclado e sem ácido.

ISBN 0201835959

17 1819202122 MA 05 04 03 02

17ª Impressão de agosto de 2002

Dedicação do 1975 edição

*A dois que enriqueceram especialmente meus anos na
IBM: Thomas J. Watson, Jr.,
cuja profunda preocupação com as pessoas ainda permeia sua
empresa, e
Bob O. Evans,
cuja ousada liderança transformou o trabalho em adventure.*

Dedicação da edição de 1995

Para Nancy,

Um presente de Deus para mim.

Prefácio à edição do 20º aniversário

Para minha surpresa e deleite, *O Mítico Homem-Mês* continua a ser popular após 20 anos. Mais de 250.000 cópias estão sendo impressas. As pessoas freqüentemente perguntam quais das opiniões e recomendações estabelecidas em 1975 eu ainda defendo, quais mudaram e como. Embora de vez em quando eu tenha abordado essa questão em palestras, há muito tempo queria escrevê-la por escrito.

Peter Gordon, agora PublishingPartner na Addison-Wesley, tem trabalhado comigo de forma paciente e prestativa desde 1980. Propus que preparássemos uma Edição de Aniversário. Decidimos não revisar o original, mas reimprimi-lo intacto (exceto para correções triviais) e aumentá-lo com pensamentos mais atuais.

O capítulo 16 reimprime "No Silver Bullet: Essence and Accidents of Software Engineering", um artigo do IFIPS de 1986 que surgiu de minha experiência como presidente de um estudo do Defense Science Board sobre software militar. Meus co-autores desse estudo e nosso secretário executivo, Robert L. Patrick, foram inestimáveis para me trazer de volta ao contato com grandes projetos de software do mundo real. O artigo foi reimpresso em 1987 no IEEE *Computador* revista, o que lhe deu ampla circulação.

"No Silver Bullet" provou ser provocativo. Ele previu que em uma década não haveria nenhuma técnica de programação que por si só trouxesse uma melhoria de ordem de magnitude na produtividade do software. A década ainda tem um ano pela frente; minha previsão parece segura. "NSB" tem estimulado discussões cada vez mais animadas

na literatura que tem *O Mítico Homem-Mês*. O Capítulo 17, portanto, comenta algumas das críticas publicadas e atualiza as opiniões apresentadas em 1986.

Ao preparar minha retrospectiva e atualização de *O Mítico Homem-Mês*, Fiquei surpreso ao ver como poucas das proposições afirmadas nele foram criticadas, comprovadas ou refutadas por pesquisas e experiências contínuas de engenharia de software. Foi útil para mim agora catalogar essas proposições em sua forma bruta, despojada de argumentos e dados de apoio. Na esperança de que essas declarações simples convidem a argumentos e fatos para provar, refutar, atualizar ou refinar essas proposições, incluí este esboço no Capítulo 18.

O capítulo 19 é o próprio ensaio de atualização. O leitor deve ser avisado de que as novas opiniões não são tão bem informadas pela experiência nas trincheiras como o livro original. Estive trabalhando em uma universidade, não na indústria, e em projetos de pequena escala, não grandes. Desde 1986, só leciono engenharia de software, não faço nenhuma pesquisa sobre o assunto. Minha pesquisa tem sido em ambientes virtuais e suas aplicações.

Ao preparar esta retrospectiva, busquei as visões atuais de amigos que estão de fato trabalhando na engenharia de software. Por uma disposição maravilhosa de compartilhar pontos de vista, comentar cuidadosamente sobre os rascunhos e me reeducar, estou em dívida com Barry Boehm, Ken Brooks, Dick Case, James Coggins, Tom DeMarco, Jim McCarthy, David Parnas, Earl Wheeler e Edward Yourdon. Fay Ward administrou soberbamente a produção técnica dos novos capítulos.

Agradeço a Gordon Bell, Bruce Buchanan, Rick Hayes-Roth, meus colegas da Força-Tarefa do Defense Science Board sobre Software Militar e, mais especialmente, David Parnas por seus insights e ideias estimulantes e Rebekah Bierly pela produção técnica do papel impresso aqui como Capítulo 16. Analisando o problema de software nas categorias de *essência* e *acidente* foi inspirado por Nancy Greenwood Brooks, que usou essa análise em um artigo sobre a pedagogia do violino Suzuki.

O costume da casa de Addison-Wesley não me permitiu reconhecer no prefácio da edição de 1975 os papéis-chave desempenhados por sua equipe. As contribuições de duas pessoas devem ser especialmente citadas: Norman Stanton, então Editor Executivo, e Herbert Boes, então Diretor de Arte. Boes desenvolveu o estilo elegante, que um revisor citou especialmente: "margens largas, [e] uso imaginativo de fonte e layout." Mais importante, ele também fez a recomendação crucial de que cada capítulo tivesse uma imagem de abertura. (Eu tinha apenas o Tar Pit e a Catedral de Reims na época.) Encontrar as fotos ocasionou um ano extra de trabalho para mim, mas sou eternamente grato pelo conselho.

Glória soli deo - Só a Deus seja a glória.

Chapel Hill, NC.

FPB, Jr.

Março de 1995

Prefácio ao Primeira edição

De muitas maneiras, gerenciar um grande projeto de programação de computador é como gerenciar qualquer outro grande empreendimento - em mais maneiras do que a maioria dos programadores acredita. Mas, de muitas outras maneiras, é diferente - em mais maneiras do que a maioria dos gerentes profissionais espera.

A tradição da área está se acumulando. Houve várias conferências, sessões em conferências AFIPS, alguns livros e artigos. Mas ainda não está em forma para qualquer tratamento sistemático de livro didático. Parece apropriado, entretanto, oferecer este pequeno livro, refletindo essencialmente uma visão pessoal.

Embora eu tenha crescido originalmente no lado da programação da ciência da computação, estive envolvido principalmente na arquitetura de hardware durante os anos (1956-1963) em que o programa de controle autônomo e o compilador de linguagem de alto nível foram desenvolvidos. Quando em 1964 me tornei gerente do Operating System / 360, descobri um mundo da programação bastante alterado pelo progresso dos anos anteriores.

Gerenciar o desenvolvimento do OS / 360 foi uma experiência muito educacional, embora muito frustrante. A equipe, incluindo F. M. Trapnell, que me sucedeu como gerente, tem muito do que se orgulhar. O sistema contém muitas excelências em design e execução, e tem sido bem-sucedido em alcançar ampla utilização. Certas ideias, principalmente de entrada-saída independente de dispositivo e gerenciamento de biblioteca externa, foram inovações técnicas

agora amplamente copiado. Agora é bastante confiável, razoavelmente eficiente e muito versátil.

O esforço não pode ser considerado totalmente bem-sucedido, no entanto. Qualquer usuário do OS / 360 percebe rapidamente como ele deve ser muito melhor. As falhas no design e na execução permeiam especialmente o programa de controle, diferentemente dos compiladores de linguagem. A maioria dessas falhas data do período de design de 1964-65 e, portanto, devem ficar sob minha responsabilidade. Além disso, o produto estava atrasado, consumia mais memória do que o planejado, os custos eram várias vezes superiores aos estimados e não funcionava muito bem até vários lançamentos após o primeiro.

Depois de deixar a IBM em 1965 para vir para Chapel Hill, conforme acordado originalmente quando assumi o OS / 360, comecei a analisar a experiência do OS / 360 para ver quais lições técnicas e de gerenciamento deveriam ser aprendidas. Em particular, eu queria explicar as experiências de gerenciamento bastante diferentes encontradas no desenvolvimento de hardware System / 360 e desenvolvimento de software OS / 360. Este livro é uma resposta tardia às sondagens de Tom Watson sobre por que a programação é difícil de gerenciar.

Nesta busca, tirei proveito de longas conversas com RP Case, gerente assistente 1964-65, e F. M. Trapnell, gerente 1965-68.1 comparei as conclusões com outros gerentes de projetos de programação jumbo, incluindo FJ Corbato de MIT, John Harr e V. Vyssotsky da Bell Telephone Laboratories, Charles Portman da International Computers Limited, AP Ershov do Laboratório de Computação da Divisão Siberiana, Academia de Ciências da URSS e AM Pietrasanta da IBM.

Minhas próprias conclusões estão incorporadas nos ensaios a seguir, que se destinam a programadores profissionais, gerentes profissionais e, especialmente, gerentes profissionais de programadores.

Embora escritos como ensaios separáveis, há um argumento central contido especialmente nos Capítulos 2-7. Resumidamente, acredito que grandes projetos de programação sofrem problemas de gerenciamento

diferentes em espécie dos pequenos, devido à divisão do trabalho. Acredito que a necessidade crítica seja a preservação da integridade conceitual do próprio produto. Estes capítulos exploram as dificuldades de se alcançar essa unidade e os métodos para fazê-lo. Os capítulos posteriores exploram outros aspectos do gerenciamento de engenharia de software.

A literatura neste campo não é abundante, mas é amplamente dispersa. Por isso, tentei fornecer referências que tanto iluminarão pontos específicos quanto guiarão o leitor interessado a outras obras úteis. Muitos amigos leram o manuscrito e alguns prepararam comentários úteis extensos; onde estes pareciam valiosos, mas não se encaixavam no fluxo do texto, eu os incluí nas notas.

Por se tratar de um livro de ensaios e não de um texto, todas as referências e notas foram banidas para o final do volume, e o leitor é instado a ignorá-las em sua primeira leitura.

Estou profundamente grato à Sra. Sara Elizabeth Moore, ao Sr. David Wagner e à Sra. Rebecca Burns por sua ajuda na preparação do manuscrito, e ao Professor Joseph C. Sloane pelos conselhos sobre ilustração.

Chapel Hill, NC

Outubro de 1974

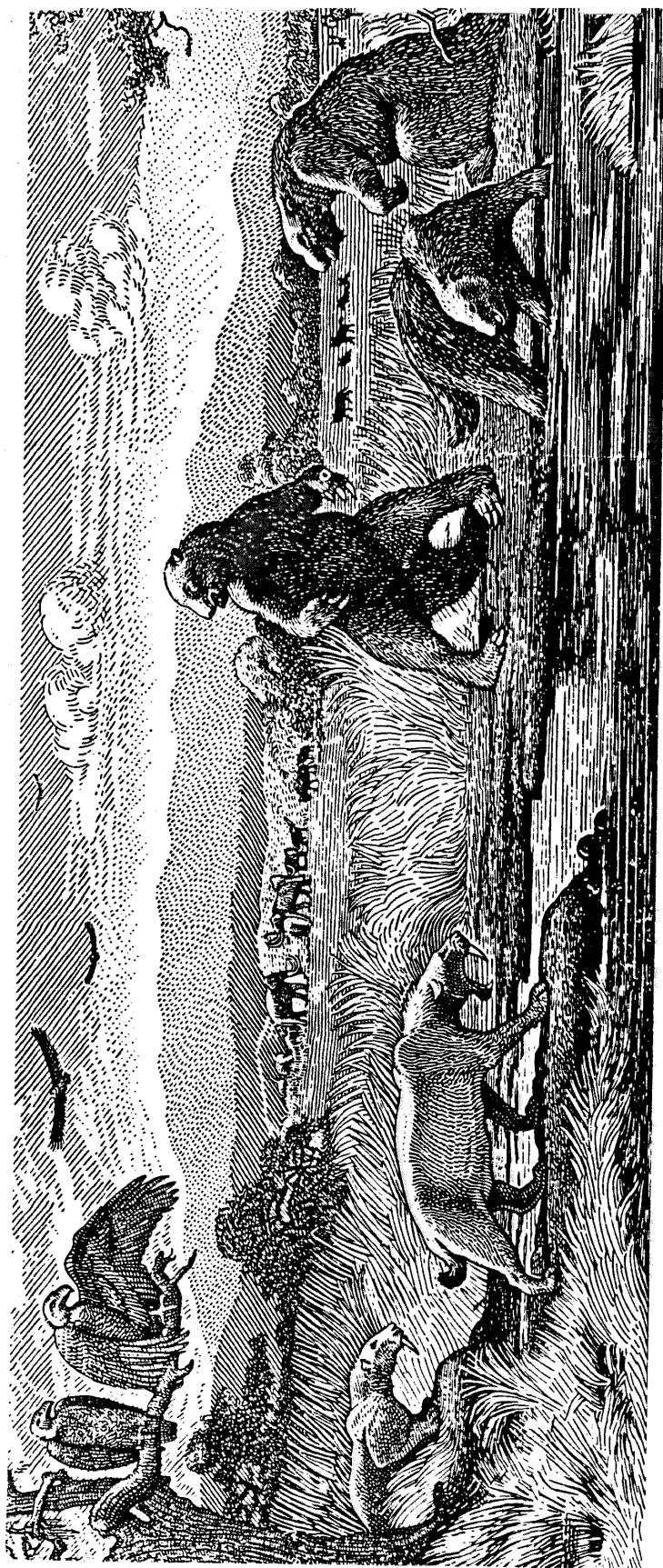
FPB, Jr

Conteúdo

Prefácio à edição do 20º aniversário	vii
Prefácio à primeira edição	x
Capítulo 1 O poço de alcatrão	3
Capítulo 2 O Mítico Homem-Mês	13
Capítulo 3 A Equipe Cirúrgica	29
Capítulo 4 Aristocracia, democracia e design do sistema O	41
capítulo 5 efeito do segundo sistema	53
Capítulo 6 Passando a Palavra	61
Capítulo 7 Por que a Torre de Babel falhou?	73
Capítulo 8 Chamando o tiro	87
Capítulo 9 Dez libras em um saco de cinco	97
Capítulo 10 libras O plano da hipótese	107
Capítulo 11 documentária para jogar fora	115
Capítulo 12 Ferramentas Sharp	127
Capítulo 13 O todo e as partes que	141
Capítulo 14 ecodem uma catástrofe	153
Capítulo 15 A outra face	163
Capítulo 16 No Silver Bullet - Essence and Accident	177
Capítulo 17 "No Silver Bullet" Refired	205
Capítulo 18 Proposições de <i>O Mítico Homem-Mês</i> :	
Verdadeiro ou falso?	227
Capítulo 19 <i>O Mítico Homem-Mês</i> após 20 anos	251
Epílogo	291
Notas e Referências	293
Índice	309

1

O poço de alcatrão



1

The Tar Pit

Een schip op het strand é een baken in zee.

{Um navio na praia é um farol para o mar.]

PROVERBIO HOLANDES

C R. Knight, mural de La Brea Tar Pits
Museu George C. Page de La Brea Discoveries, Museu
de História Natural do Condado de Los Angeles

Nenhuma cena da pré-história é tão vívida como a das lutas mortais de grandes feras nos poços de piche. No olho da mente, vemos dinossauros, mamutes e tigres com dentes de sabre lutando contra o alcatrão. Quanto mais feroz a luta, mais emaranhado o alcatrão, e nenhum animal é tão forte ou tão hábil a não ser afundar.

A programação de grandes sistemas tem sido na última década um poço de alcatrão, e muitas feras grandes e poderosas se debateram violentamente com isso. A maioria surgiu com sistemas em execução - poucos cumpriram metas, cronogramas e orçamentos. Grandes e pequenos, maciços ou ríjos, time após time se enredam no alcatrão. Nada parece causar a dificuldade - qualquer pata em particular pode ser puxada. Mas o acúmulo de fatores simultâneos e interagentes traz um movimento cada vez mais lento. Todos parecem ter ficado surpresos com a persistência do problema e é difícil discernir sua natureza. Mas devemos tentar entendê-lo se quisermos resolvê-lo.

Portanto, vamos começar identificando a arte da programação do sistema e as alegrias e tristezas inerentes a ela.

O Produto de Sistemas de Programação

Ocasionalmente, lê-se relatos de jornal sobre como dois programadores em uma garagem reformada construíram um programa importante que supera os melhores esforços de grandes equipes. E todo programador está preparado para acreditar em tais histórias, pois ele sabe que poderia construir *algum* programa muito mais rápido do que as 1000 declarações / ano relatadas para equipes industriais.

Por que então nem todas as equipes de programação industrial foram substituídas por duplas de garagem dedicadas? É preciso olhar para *That* está sendo produzido.

No canto superior esquerdo da Fig. 1.1 é um *programa*. Ele é completo em si mesmo, pronto para ser executado pelo autor no sistema no qual foi desenvolvido. *Este* é a coisa comumente produzida em garagens, e

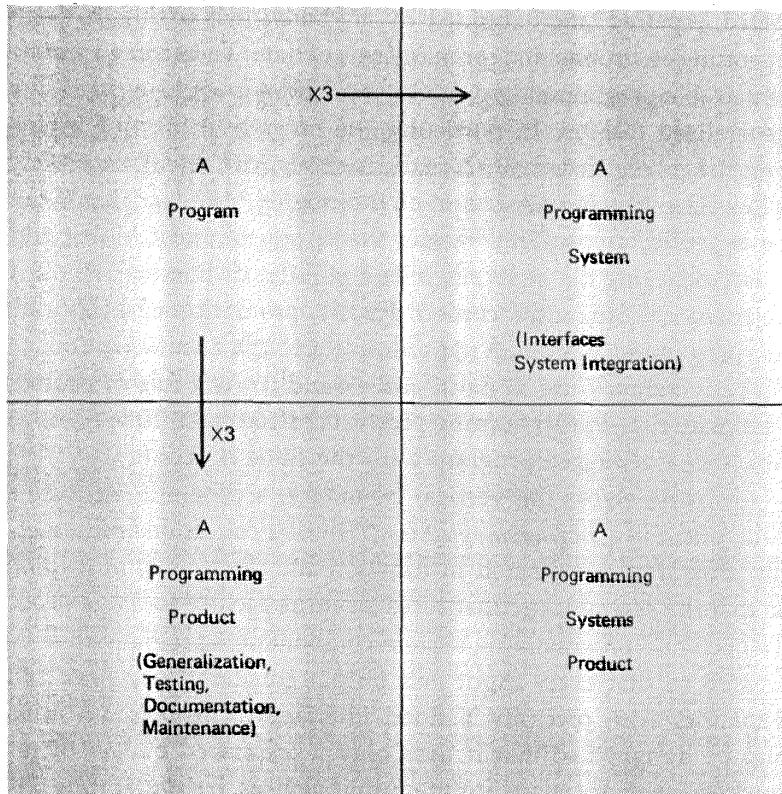


Fig. 1.1 Evolução do produto de sistemas de programação

esse é o objeto que o programador individual usa para estimar a produtividade.

Existem duas maneiras de converter um programa em um objeto mais útil, porém mais caro. Essas duas maneiras são representadas pelos limites no diagrama.

Descendo pelo limite horizontal, um programa se torna um *produto de programação*. Este é um programa que pode ser executado,

testado, reparado e ampliado por qualquer pessoa. É utilizável em muitos ambientes operacionais, conjuntos de dados formay. Para se tornar um produto de programação utilizable em geral, um programa deve ser escrito de maneira generalizada. Em particular, o intervalo e a forma das entradas devem ser generalizados tanto quanto o algoritmo básico permitir. Em seguida, o programa deve ser completamente testado, para que se possa confiar nele. Isso significa que um banco substancial de casos de teste, explorando a faixa de entrada e sondando seus limites, deve ser preparado, executado e registrado. Finalmente, a promoção de um programa a produto de programação requer sua documentação completa, para que qualquer pessoa possa usá-la, corrigi-la e estendê-la. Como regra geral, estimo que um produto de programação custe pelo menos três vezes mais que um programa depurado com a mesma função.

Movendo-se através da fronteira vertical, um programa se torna um componente em *sistema de programação*. Trata-se de uma coleção de programas em interação, coordenados em função e disciplinados em formato, de modo que o conjunto constitua toda uma facilidade para grandes tarefas. Para se tornar um componente do sistema de programação, um programa deve ser escrito de forma que cada entrada e saída esteja em conformidade com a sintaxe e semântica com interfaces precisamente definidas. O programa também deve ser projetado de forma que use apenas um determinado orçamento de recursos - espaço de memória, dispositivos de entrada-saída, tempo de computador. Finalmente, o programa deve ser testado com outros componentes do sistema, em todas as combinações esperadas. Esse teste deve ser extenso, pois o número de casos cresce combinatoriamente. É demorado, pois bugs sutis surgem de interações inesperadas de componentes depurados. Um componente do sistema de programação custa pelo menos três vezes mais que um programa independente da mesma função.

No canto inferior direito da Fig. 1.1 está o *produto de sistemas de programação*. Isso difere do programa simples em todas as maneiras acima. Custa nove vezes mais. Mas é o objeto verdadeiramente útil, o produto pretendido da maioria dos esforços de programação de sistema.

As alegrias da arte

Por que programar é divertido? Que delícias seu praticante pode esperar como recompensa?

O primeiro é a pura alegria de fazer coisas. Assim como a criança adora sua torta de lama, o adulto gosta de construir coisas, especialmente as que ele mesmo criou. Acho que esse deleite deve ser uma imagem do deleite de Deus em fazer as coisas, um deleite mostrado na distinção e novidade de cada folha e cada floco de neve.

Em segundo lugar, está o prazer de fazer coisas que sejam úteis para outras pessoas. No íntimo, queremos que outros usem nosso trabalho e o considerem útil. Nesse aspecto, o sistema de programação não é essencialmente diferente do primeiro porta-lápis de argila da criança "para o escritório do papai".

Em terceiro lugar, está o fascínio de criar objetos semelhantes a quebra-cabeças complexos de peças móveis interligadas e observá-las trabalhar em ciclos sutis, desempenhando as consequências de princípios construídos desde o início. O computador programado tem todo o fascínio da máquina de pinball ou do mecanismo de jukebox, levado ao máximo.

Em quarto lugar, está a alegria de aprender sempre, que surge da natureza não repetitiva da tarefa. De uma forma ou de outra, o problema é sempre novo e seu solucionador aprende algo: às vezes prático, às vezes teórico e às vezes ambos.

Finalmente, há o prazer de trabalhar em um meio tão tratável. O programador, como o poeta, trabalha apenas levemente afastado do puro material de pensamento. Ele constrói seus castelos no ar, do ar, criando pelo esforço da imaginação. Poucos meios de criação são tão flexíveis, tão fáceis de polir e retrabalhar, tão prontamente capazes de realizar grandes estruturas conceituais. (Como veremos mais tarde, essa mesma tratabilidade tem seus próprios problemas.)

No entanto, a construção do programa, ao contrário das palavras do poeta, é real no sentido de que se move e funciona, produzindo saídas visíveis separadas da própria construção. Imprime resultados, desenha, produz sons, movimenta braços. A magia do mito e da lenda tem

se tornou realidade em nosso tempo. Digita-se o encantamento correto no teclado, e uma tela ganha vida, mostrando coisas que nunca existiram nem poderiam ser.

A programação, então, é divertida porque gratifica anseios criativos construídos profundamente dentro de nós e encanta as sensibilidades que temos em comum com todos os homens.

As desgraças do ofício

Nem tudo é deleite, porém, e conhecer os infortúnios inerentes torna mais fácil suportá-los quando aparecem.

Primeiro, é preciso ter um desempenho perfeito. O computador também se assemelha à magia da lenda nesse aspecto. Se um personagem, uma pausa, do encantamento não estiver estritamente na forma adequada, a magia não funciona. Os seres humanos não estão acostumados a ser perfeitos e poucas áreas da atividade humana o exigem. Ajustar-se ao requisito de perfeição é, penso eu, a parte mais difícil de aprender a programar.¹

Em seguida, outras pessoas definem seus objetivos, fornecem seus recursos e fornecem suas informações. Raramente se controla as circunstâncias de seu trabalho, ou mesmo seu objetivo. Em termos de gestão, a autoridade de uma pessoa não é suficiente para sua responsabilidade. Parece que em todos os campos, no entanto, os trabalhos em que as coisas são feitas nunca têm autoridade formal proporcional à responsabilidade. Na prática, a autoridade real (em oposição à formal) é adquirida desde o momento da realização.

A dependência de outros tem um caso particular que é especialmente doloroso para o programador de sistema. Ele depende dos programas de outras pessoas. Muitas vezes, são mal planejados, mal implementados, entregues de forma incompleta (sem código-fonte ou casos de teste) e mal documentados. Portanto, ele deve passar horas estudando e consertando coisas que em um mundo ideal seriam completas, disponíveis e utilizáveis.

A próxima desgraça é que projetar grandes conceitos é divertido; encontrar pequenos bugs é apenas trabalho. Com qualquer atividade criativa, venha

horas enfadonhas de trabalho tedioso e trabalhoso e programação não é exceção.

Em seguida, descobre-se que a depuração tem uma convergência linear, ou pior, onde se espera de alguma forma uma espécie de abordagem quadrática para o fim. Portanto, os testes se arrastam indefinidamente, os últimos bugs difíceis demorando mais para serem encontrados do que o primeiro.

A última desgraça, e às vezes a gota d'água, é que o produto sobre o qual se trabalhou por tanto tempo parece estar obsoleto após (ou antes) da conclusão. Os colegas e concorrentes já estão em busca de ideias novas e melhores. Já o deslocamento do filho pensante não é apenas concebido, mas programado.

Isso sempre parece pior do que realmente é. O produto novo e melhor geralmente não é *acessível*/quando um completa o seu; é apenas falado. Também exigirá meses de desenvolvimento. O tigre real nunca é páreo para o de papel, a menos que o uso real seja desejado. Então, as virtudes da realidade têm uma satisfação própria.

Claro que a base tecnológica sobre a qual se constrói é *sempre* avançando. Assim que se congela um design, ele se torna obsoleto em termos de seus conceitos. Mas a implementação de produtos reais exige faseamento e quantização. A obsolescência de uma implementação deve ser medida em relação a outras implementações existentes, não a conceitos não realizados. O desafio e a missão são encontrar soluções reais para problemas reais nos cronogramas atuais com os recursos disponíveis.

Isso então é programação, tanto um poço de piche em que muitos esforços fracassaram quanto uma atividade criativa com alegrias e infortúnios próprios. Para muitos, as alegrias superam em muito as angústias e, para eles, o restante deste livro tentará colocar alguns calçadões sobre o alcatrão.

2

O Mítico Homem-Mês

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	Entrecôte marchand de vin	4.00
Côtelettes d'agneau aux champignons frais	2.75	Côtelettes d'agneau maison d'or	2.
Filet de boeuf aux champignons frais	4.75	Côtelettes d'agneau à la parisienne	2.
Ris de veau à la financière	2.00	Fois de volaille à la brochette	1.50
Filet de boeuf nature	3.75	Tournedos nature	2.75
Tournedos Médicis	3.25	Filet de boeuf à la hawaïenne	4.00
Pigeonneaux sauce paradis	3.50	Tournedos à la hawaïenne	3.25
Tournedos sauce béarnaise	3.25	Tournedos marchand de vin	3.25
Entrecôte minute	2.75	Pigeonneaux grillés	3.00
Filet de boeuf béarnaise	4.00	Entrecôte nature	3.75
Tripes à la mode de Caen (commander d'avance)	2.00	Châteaubriand (30 minutes)	

LÉGUMES

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

SALADES

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

DESSERTS

Gâteau moka	.50	Cerises jubilé	1.25
Meringue glacée	.60	Crêpes à la gelée	.80
Crêpes Suzette	1.25	Crêpes nature	.70
Glace sauce chocolat	.60	Omelette au rhum	1.10
Fruits de saison à l'eau-de-vie	.75	Glace à la vanille	.5
Omelette soufflée à la Jules César (2)	2.00	Fraises au kirch	
Omelette Alaska Antoine (2)	2.50	Pêche Melba	

FROMAGES

Roquefort	.50	Liederkranz	.50
Camembert	.50	Gruyère	.50

Fromage à la crème Philadelphie .50

CAFÉ ET THÉ

Café	.20	Café au lait	.20
Café brûlé diabolique	1.00	Thé glacé	.20

Thé .20

Demi-tasse

EAUX MINÉRALES — BIÈRE — CIGARES — CIGARETTES

White Rock	Bière locale	Cigares
Vichy	Cliquot Club	Canada Dry

Cig

Cigarettes



Roy L. Alciatore, Propriétaire

713-717 Rue St. Louis

Nouvelle Orléans, Louisiane

2

O Mítico Homem-Mês

Uma boa cozinha finge tempo. Se você é feito para esperar, é para servi-lo melhor e para agradá-lo.

MENU DO RESTAURANTE ANTOINE. NOVA ORLEANS

14 O Mítico Homem-Mês

Mais projetos de software deram errado por falta de tempo no calendário do que por todas as outras causas combinadas. Por que essa causa de desastre é tão comum?

Primeiro, nossas técnicas de estimativa são mal desenvolvidas. Mais seriamente, eles refletem uma suposição não expressa que é totalmente falsa, isto é, que tudo correrá bem.

Em segundo lugar, nossas técnicas de estimativa confundem falaciosamente esforço com progresso, escondendo a suposição de que homens e meses são intercambiáveis.

Terceiro, como não temos certeza de nossas estimativas, os gerentes de software geralmente carecem da teimosia cortês do chef de Antoine.

Quarto, o progresso do cronograma é mal monitorado. Técnicas comprovadas e rotineiras em outras disciplinas da engenharia são consideradas inovações radicais na engenharia de software.

Quinto, quando o atraso no cronograma é reconhecido, a resposta natural (e tradicional) é adicionar mão de obra. Como apagar um incêndio com gasolina, isso torna as coisas piores, muito piores. Mais fogo requer mais gasolina e, portanto, começa um ciclo regenerativo que termina em desastre.

O monitoramento do cronograma será objeto de um ensaio à parte. Vamos considerar outros aspectos do problema com mais detalhes.

Otimismo

Todos os programadores são otimistas. Talvez essa feitiçaria moderna atraia especialmente aqueles que acreditam em finais felizes e fadas madrinhas. Talvez as centenas de frustrações essenciais afastem todos, exceto aqueles que habitualmente se concentram no objetivo final. Talvez seja apenas porque os computadores são jovens, os programadores são mais jovens e os jovens são sempre otimistas. Mas como quer que o processo de seleção funcione, o resultado é indiscutível: "Desta vez com certeza vai rodar" ou "Acabei de encontrar o último bug."

Portanto, a primeira falsa suposição subjacente à programação da programação de sistemas é que *tudo vai correr bem*, ou seja, isso *cada tarefa irá caminhe apenas o tempo que "deveria" durar*.

A difusão do otimismo entre os programadores merece mais do que uma análise inversa. Dorothy Sayers, em seu excelente livro, *A mente do criador*, divide a atividade criativa em três estágios: a ideia, a implementação e a interação. Um livro, então, ou um computador, ou um programa passa a existir primeiro como uma construção ideal, construída fora do tempo e do espaço, mas completa na mente do autor. É realizado no tempo e no espaço, por caneta, tinta e papel, ou por arame, silício e ferrite. A criação é concluída quando alguém lê o livro, usa o computador ou executa o programa, interagindo assim com a mente do criador.

Esta descrição, que a Srta. Sayers usa para iluminar não apenas a atividade criativa humana, mas também a doutrina cristã da Trindade, nos ajudará em nossa tarefa presente. Para os criadores humanos das coisas, as lacunas e inconsistências de nossas idéias tornam-se claras apenas durante a implementação. É assim que escrever, experimentar, "trabalhar fora" são disciplinas essenciais para o teórico.

Em muitas atividades criativas, o meio de execução é intratável. Lumber splits; manchas de tinta; anel de circuitos elétricos. Essas limitações físicas do meio restringem as idéias que podem ser expressas e também criam dificuldades inesperadas na implementação.

A implementação, então, leva tempo e suor por causa da mídia física e por causa da inadequação das ideias subjacentes. Temos a tendência de culpar a mídia física pela maioria de nossas dificuldades de implementação; pois a mídia não é "nossa" como as idéias são, e nosso orgulho influencia nosso julgamento.

A programação de computador, no entanto, cria um meio extremamente tratável. O programador constrói a partir de puro pensamento: conceitos e representações muito flexíveis dos mesmos. Como o meio é tratável, esperamos poucas dificuldades na implementação; daí o nosso otimismo generalizado. Porque nossas ideias são falhas, temos bugs; portanto, nosso otimismo é injustificado.

Em uma única tarefa, a suposição de que tudo correrá bem tem um efeito probabilístico no cronograma. Pode de fato funcionar como **planned**.

16 O Mítico Homem-Mês

pois há uma distribuição de probabilidade para o atraso que será encontrado, e "sem atraso" tem uma probabilidade finita. Um grande esforço de programação, no entanto, consiste em muitas tarefas, algumas encadeadas de ponta a ponta. A probabilidade de que cada um dê certo torna-se extremamente pequena.

The'Man-Month

O segundo modo de pensamento falacioso é expresso na própria unidade de esforço usada na estimativa e na programação: o homem-mês. O custo realmente varia conforme o produto do número de homens e do número de meses. O progresso não. *Portanto, o homem-mês como unidade para medir o tamanho de um trabalho é um mito perigoso e enganoso.* Isso implica que homens e meses são intercambiáveis.

Homens e meses são mercadorias intercambiáveis apenas quando uma tarefa pode ser repartida entre muitos trabalhadores *com no comunição entre eles* (Fig. 2.1). Isso vale para a colheita do trigo ou do algodão; não é nem mesmo aproximadamente verdadeiro para a programação de sistemas.

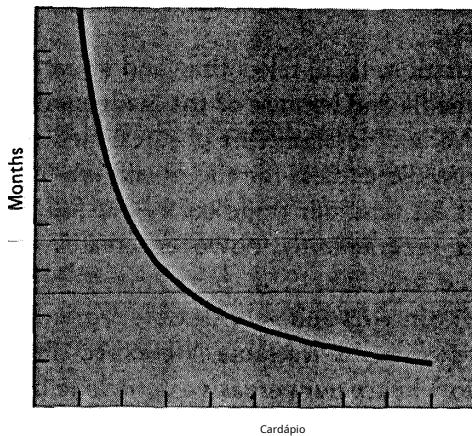


Fig. 2.1 Tempo versus número de trabalhadores - tarefa perfeitamente partionável

Quando uma tarefa não pode ser particionada devido a restrições sequenciais, a aplicação de mais esforço não tem efeito sobre o cronograma (Fig. 2.2). O parto de um filho leva nove meses, não importa quantas mulheres sejam designadas. Muitas tarefas de software têm essa característica devido à natureza sequencial da depuração.

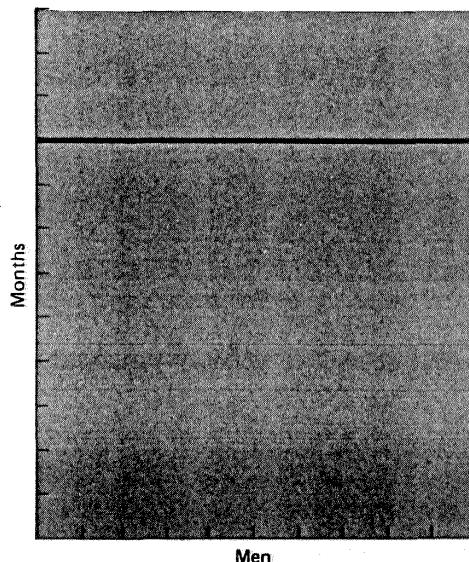


Fig. 2.2 Tempo versus número de trabalhadores - tarefa não particionável

Em tarefas que podem ser particionadas, mas que requerem comunicação entre as subtarefas, o esforço de comunicação deve ser adicionado à quantidade de trabalho a ser realizado. Portanto, o melhor que pode ser feito é um pouco mais pobre do que um comércio uniforme de homens durante meses (Fig. 2.3).

18 O Mítico Homem-Mês

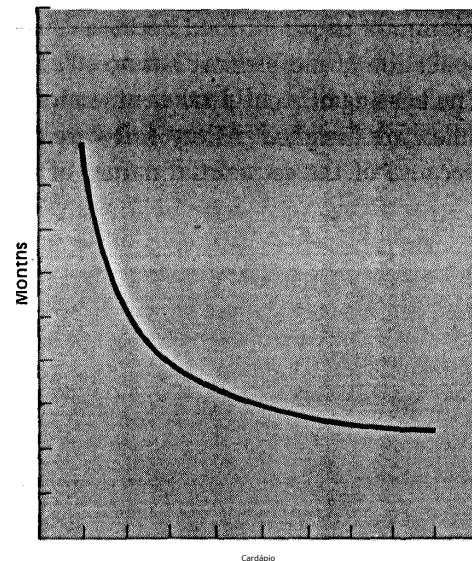


Fig. 2.3 Tempo versus número de trabalhadores - tarefa particionável que requer comunicação

A carga adicional de comunicação é composta de duas partes, treinamento e intercomunicação. Cada trabalhador deve ser treinado na tecnologia, nos objetivos do esforço, na estratégia geral e no plano de trabalho. Este treinamento não pode ser dividido, portanto, essa parte do esforço adicionado varia linearmente com o número de trabalhadores.¹

A intercomunicação é pior. Se cada parte da tarefa deve ser coordenada separadamente uma com a outra parte / o esforço aumenta à medida que $n(nI)/2$. Três trabalhadores requerem três vezes mais intercomunicação em pares do que dois; quatro requerem seis vezes mais que dois. Se, além disso, houver necessidade de conferências entre três, quatro, etc., os trabalhadores para resolverem as coisas em conjunto, as coisas pioram ainda. O esforço adicional de comunicação pode neutralizar totalmente a divisão da tarefa original e nos levar à situação da Figura 2.4.

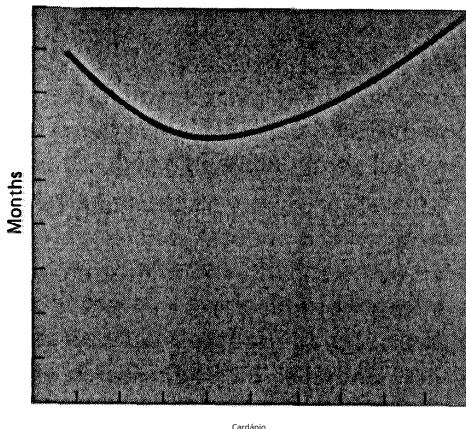


FIG. 2.4 Tempo versus número de trabalhadores - tarefa com inter-relacionamentos complexos

Uma vez que a construção de software é inherentemente um esforço de sistema - um exercício em inter-relacionamentos complexos - o esforço de comunicação é grande e rapidamente domina a diminuição do tempo de tarefa individual ocasionada pelo particionamento. Adicionar mais homens aumenta, não encurta, o cronograma.

Sistemas Teste

Nenhuma parte do cronograma é tão afetada por restrições sequenciais como depuração de componentes e teste de sistema. Além disso, o tempo necessário depende do número e da sutileza dos erros encontrados. Teoricamente, esse número deveria ser zero. Por causa do otimismo, normalmente esperamos que o número de bugs seja

menor do que acaba sendo. Portanto, o teste geralmente é a parte mais mal programada da programação.

Por alguns anos, tenho usado com sucesso a seguinte regra prática para agendar uma tarefa de software:

eu / 3 planejamento

eu / 6 codificação

eu / 4 teste de componentes e teste inicial do sistema

eu / 4 teste do sistema, todos os componentes em mãos.

Isso difere da programação convencional de várias maneiras importantes:

1. A fração dedicada ao planejamento é maior do que o normal. Mesmo assim, é apenas o suficiente para produzir uma especificação detalhada e sólida, e não o suficiente para incluir a pesquisa ou exploração de técnicas totalmente novas.
2. O *metade* da programação dedicada à depuração do código concluído é muito maior do que o normal.
3. A parte que é fácil de estimar, ou seja, a codificação, recebe apenas um sexto da programação.

Ao examinar projetos convencionalmente programados, descobri que poucos permitiam que metade do cronograma projetado fosse testado, mas a maioria realmente gastou metade do cronograma real para esse propósito. Muitos deles estavam dentro do cronograma até e exceto nos testes do sistema.²

Deixar de permitir tempo suficiente para o teste do sistema, em particular, é particularmente desastroso. Uma vez que o atraso ocorre no final do cronograma, ninguém sabe do problema com o cronograma até quase a data de entrega. Mais notícias, tardias e sem aviso prévio, são inquietantes para clientes e gerentes.

Além disso, o atraso neste ponto tem repercussões financeiras e psicológicas invulgarmente graves. O projeto está totalmente equipado e o custo por dia é máximo. Mais seriamente, o software deve apoiar outros esforços de negócios (remessa de computadores, operação de novas instalações, etc.) e os custos secundários de atrasá-los são muito altos, pois está quase na hora da remessa de software.

Regenerativo Agendar desastre 21

Na verdade, esses custos secundários podem superar em muito todos os outros. Portanto, é muito importante permitir tempo suficiente de teste do sistema na programação original.

Estimativa Gutless

Observe que para o programador, assim como para o chef, a urgência do patrono pode reger a realização programada da tarefa, mas não pode reger a realização efetiva. Uma omelete, prometida em dois minutos, pode parecer estar progredindo bem. Mas, quando não endurece em dois minutos, o cliente tem duas opções - esperar ou comer cru. Os clientes de software tiveram as mesmas opções.

O cozinheiro tem outra escolha; ele pode aumentar o calor. O resultado costuma ser uma omelete que nada pode salvar - queimada em uma parte e crua na outra.

Bem, não acho que os gerentes de software tenham menos coragem e firmeza inerentes do que os chefs, nem do que outros gerentes de engenharia. Mas a programação falsa para coincidir com a data desejada do patrono é muito mais comum em nossa disciplina do que em qualquer outro lugar da engenharia. É muito difícil fazer uma defesa vigorosa, plausível e arriscada de uma estimativa que não é derivada por nenhum método quantitativo, apoiada por poucos dados e certificada principalmente pelos palpites dos gerentes.

Claramente, duas soluções são necessárias. Precisamos desenvolver e divulgar números de produtividade, números de incidência de bugs, regras de estimativa e assim por diante. Toda a profissão só pode lucrar com o compartilhamento de tais dados.

Até que a estimativa esteja em uma base mais sólida, os gerentes individuais precisarão endurecer sua espinha dorsal e defender suas estimativas com a garantia de que seus maus palpites são melhores do que estimativas desejadas.

Desastre de cronograma regenerativo

O que fazer quando um projeto de software essencial está atrasado? Adicione força de trabalho, naturalmente. Como as Figs. 2.1 a 2.4 sugerem, isso pode ou não ajudar.

Vamos considerar um exemplo. Suponha que uma tarefa seja estimada em 12 homens-mês e atribuída a três homens por quatro meses, e que haja marcos mensuráveis A, B, C, D, que estão programados para cair no final de cada mês (Fig. 2.5).

Agora, suponha que a primeira milha não seja alcançada antes de decorridos dois meses (Fig. 2.6). Quais são as alternativas que o gerente enfrenta?

1. Suponha que a tarefa deve ser realizada no prazo. Suponha que apenas a primeira parte da tarefa foi mal avaliada, então a Fig. 2.6 conta a história com precisão. Então, restam 9 homens-mês de esforço, e mais dois meses, de modo que serão necessários 4 homens-mês. Adicione 2 homens aos 3 designados.
2. Suponha que a tarefa deve ser realizada no prazo. Suponha que toda a estimativa foi uniformemente baixa, de modo que a Fig. 2.7 realmente descreve a situação. Então, restam 18 homens-mês de esforço, e mais dois meses, portanto, serão necessários 9 homens. Adicione 6 homens aos 3 designados.

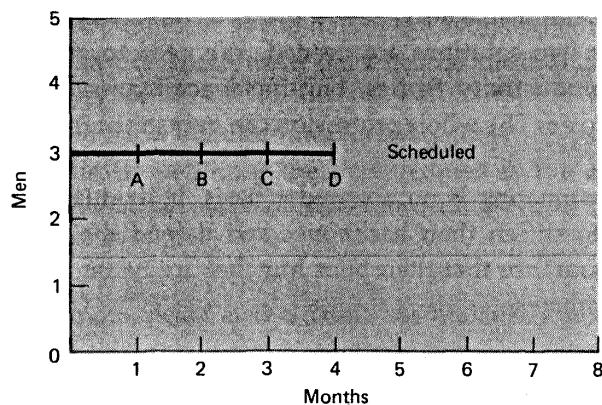


Figura 2.5

Cronograma Regenerativo Desastre

2,3

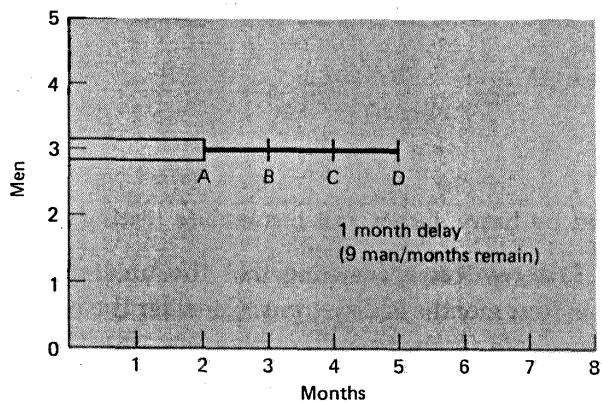


Figura 2.6

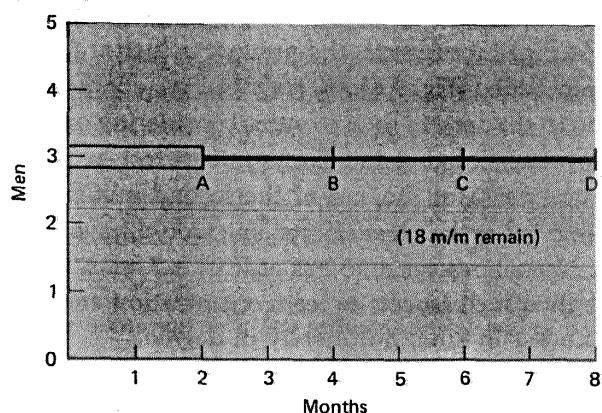


Figura 2.7

24 O Mítico Homem-Mês

3. Reprogramar. Gosto do conselho dado por P. Fagg, um engenheiro de hardware experiente, "Não dê pequenos deslizes." Ou seja, reserve tempo suficiente no novo cronograma para garantir que o trabalho possa ser feito de forma cuidadosa e completa e que a reprogramação não precise ser feita novamente.
4. Corte a tarefa. Na prática, isso tende a acontecer de qualquer maneira, uma vez que a equipe observa o desvio de cronograma. Onde os custos secundários do atraso são muito altos, esta é a única ação viável. As únicas alternativas do gerente são apará-lo formalmente e com cuidado, reprogramar ou assistir a tarefa ser silenciosamente aparada por um projeto apressado e testes incompletos.

Nos primeiros dois casos, insistir que a tarefa inalterada seja concluída em quatro meses é desastroso. Considere os efeitos regenerativos, por exemplo, para a primeira alternativa (Fig. 2.8). Os dois novos homens, por mais competentes e rapidamente recrutados, exigirão treinamento na tarefa por um dos homens experientes. Se isso levar um mês, *3 homens-mês terão sido dedicados ao trabalho que não está na estimativa original*. Além disso, a tarefa, originalmente partionada de três maneiras, deve ser reparticionada em cinco partes; portanto, algum trabalho já feito será perdido e os testes do sistema devem ser prolongados. Portanto, no final do terceiro mês, restam substancialmente mais de 7 homens-mês de esforço e 5 pessoas treinadas e um mês estão disponíveis. Como a Fig. 2.8 sugere, o produto é tão atrasado como se ninguém tivesse sido adicionado (Fig. 2.6).

Para ter a esperança de terminar em quatro meses, considerando apenas o tempo de treinamento e não o reparticionamento e teste de sistemas extras, seria necessário adicionar 4 homens, não 2, no final do segundo mês. Para cobrir o reparticionamento e os efeitos do teste do sistema, seria necessário adicionar outros homens. Agora, entretanto, um tem pelo menos uma equipe de 7 homens, não uma equipe de 3 homens; assim, aspectos como organização da equipe e divisão de tarefas são diferentes em tipo, não apenas em grau.

Observe que no final do terceiro mês as coisas parecem muito pretas. O marco de 1º de março não foi alcançado, apesar de todos

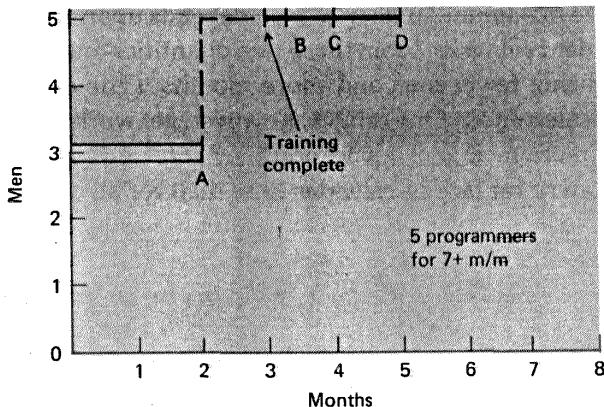


Figura 2.8

o esforço gerencial. É muito forte a tentação de repetir o ciclo, acrescentando ainda mais mão de obra. É aí que reside a loucura.

O anterior presumiu que apenas o primeiro marco foi mal estimado. Se em março *eu* faz-se a suposição conservadora de que todo o cronograma era otimista, como mostra a Fig. 2.7, queremos adicionar 6 homens apenas para a tarefa original. O cálculo dos efeitos do treinamento, reparticionamento e teste do sistema é deixado como um exercício para o leitor. Sem dúvida, o desastre regenerativo renderá um produto pior, mais tarde, do que seria reprogramado com os três homens originais, sem aumento.

Para simplificar demais, declaramos a Lei de Brooks:

Adicionar mão de obra a um projeto de software atrasado o torna mais tarde.

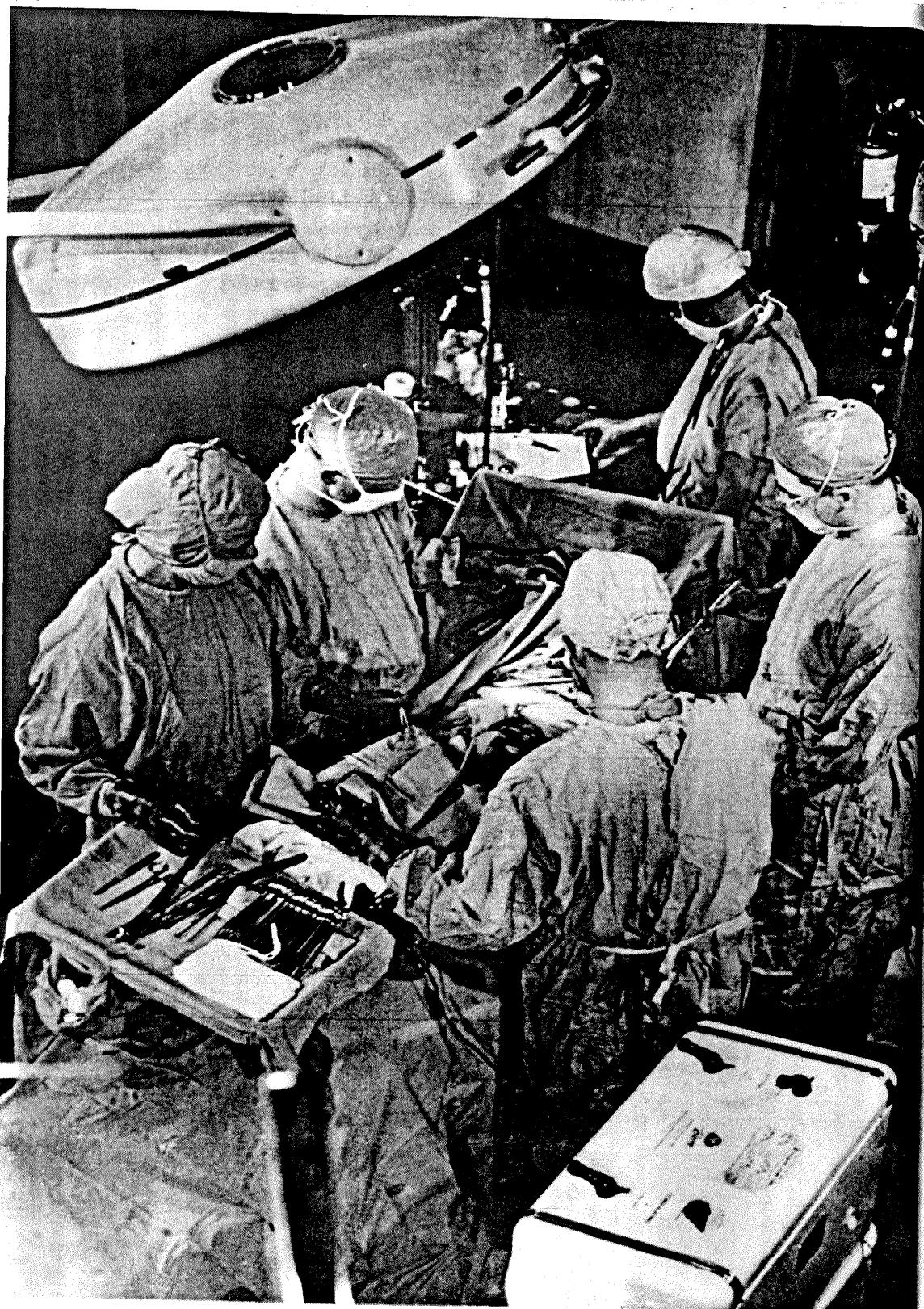
Esta então é a desmitologização do homem-mês. O número de meses de um projeto depende de sua consequência sequencial

26 O Mítico Homem-Mês

tensões. O número máximo de homens depende do número de subtarefas independentes. Destas duas quantidades, pode-se derivar horários usando menos homens e mais meses. (O único risco é a obsolescência do produto.) Não se pode, entretanto, conseguir horários viáveis usando mais homens e menos meses. Mais projetos de software deram errado por falta de tempo no calendário do que por todas as outras causas combinadas.

3

A Equipe Cirúrgica



3

A Equipe Cirúrgica

Esses estudos revelaram grandes diferenças individuais entre os de alto e baixo desempenho, geralmente em uma ordem de magnitude.

SACKMAN, ERIKSON, E CONCESSÃO,

Foto UPI / Arquivo Bettman

30 A Equipe Cirúrgica

Nas reuniões da sociedade da computação, ouve-se continuamente jovens gerentes de programação afirmarem que favorecem uma equipe pequena e afiada de pessoas de primeira classe, em vez de um projeto com centenas de programadores, e aqueles por implicação medíocre. Todos nós também.

Mas esta afirmação ingênua das alternativas evita o difícil problema - como alguém constrói *ampla* sistemas em uma programação significativa? Vejamos cada lado dessa questão com mais detalhes.

O problema

Os gerentes de programação há muito tempo reconhecem grandes variações de produtividade entre bons e pobres programadores. Mas as magnitudes reais medidas surpreenderam a todos nós. Em um de seus estudos, Sackman, Erikson e Grant estavam medindo o desempenho de um grupo de programadores experientes. Apenas neste grupo, as relações entre o melhor e o pior desempenho foram em média de cerca de 10: 1 nas medições de produtividade e incríveis 5: 1 nas medições de velocidade e espaço do programa! Resumindo, o programador de \$ 20.000 / ano pode muito bem ser 10 vezes mais produtivo do que o de \$ 10.000 / ano. O contrário também pode ser verdade. Os dados não mostraram nenhuma correlação entre experiência e desempenho.

Argumentei anteriormente que o grande número de mentes a serem coordenadas afeta o custo do esforço, pois uma grande parte do custo é a comunicação e a correção dos efeitos nocivos da falta de comunicação (depuração do sistema). Isso também sugere que se deseja que o sistema seja construído pelo menor número possível de mentes. De fato, a maior parte da experiência com grandes sistemas de programação mostra que a abordagem de força bruta é cara, lenta, ineficiente e produz sistemas que não são conceitualmente integrados. OS / 360, Exec 8, Scope 6600, Multics, TSS, SAGE, etc. - a lista é infinita.

A conclusão é simples: se um projeto de 200 homens tem 25 gerentes que são os programadores mais competentes e experientes, dispare 175 soldados e coloque os gerentes de volta na programação.

Agora vamos examinar essa solução. Por um lado, não consegue se aproximar do ideal da *pequena equipe afiada*, que por consenso comum não deve exceder 10 pessoas. É tão grande que precisará ter pelo menos dois níveis de gestão, ou cerca de cinco gerentes. Além disso, precisará de apoio em finanças, pessoal, espaço, secretárias e operadores de máquinas.

Por outro lado, a equipe original de 200 homens não era grande o suficiente para construir sistemas realmente grandes por métodos de força bruta. Considere o OS / 360, por exemplo. No pico, mais de 1.000 pessoas estavam trabalhando nisso - programadores, escritores, operadores de máquinas, funcionários, secretárias, gerentes, grupos de apoio e assim por diante. De 1963 a 1966, provavelmente 5.000 homens-ano foram gastos em seu projeto, construção e documentação. Nossa equipe postulada de 200 homens levaria 25 anos para trazer o produto ao seu estágio atual, se homens e meses fossem negociados uniformemente!

Este é então o problema com o conceito de equipe pequena e precisa: é *muito lento para sistemas realmente grandes*. Considere o trabalho do OS / 360, pois ele pode ser resolvido por uma equipe pequena e afiada. Inscreva-se para uma equipe de 10 homens. Como limite, deixe-os ser sete vezes mais produtivos que os programadores medíocres tanto em programação quanto em documentação, porque são afiados. Suponha que OS / 360 foi construído apenas por programadores medíocres (que é *longe* da verdade). Como limite, suponha que outro fator de melhoria de produtividade de sete venha da comunicação reduzida por parte da equipe menor. Suponha que *mesmo* a equipe permanece no trabalho inteiro. Bem, $5000 / (10 \times 7 \times 7) = 10$; eles podem fazer o trabalho de 5.000 homens-ano em 10 anos. O produto será interessante 10 anos após seu design inicial? Ou terá se tornado obsoleto devido ao rápido desenvolvimento da tecnologia de software?

O dilema é cruel. Para eficiência e integridade conceitual, prefere-se alguns bons pensamentos fazendo design e construção. Ainda assim, para grandes sistemas, deseja-se uma forma de trazer mão de obra considerável para suportar, de modo que o produto possa aparecer em tempo hábil. Como essas duas necessidades podem ser reconciliadas?

Proposta de Mills

Uma proposta de Harlan Mills oferece uma solução nova e criativa.²³ Mills propõe que cada segmento de um grande trabalho seja tratado por uma equipe, mas que a equipe seja organizada como uma equipe cirúrgica, em vez de uma equipe de abate de porcos. Ou seja, ao invés de cada membro cortar o problema, um faz o corte e os outros dão a ele todo o suporte que irá aumentar sua eficácia e produtividade.

Um pouco de reflexão mostra que esse conceito atende aos desideratos, se for possível fazê-lo funcionar. Poucas mentes estão envolvidas no projeto e na construção, mas muitas mãos são postas em prática. Isso pode funcionar? Quem são os anestesiologistas e enfermeiras da equipe de programação e como é dividido o trabalho? Deixe-me misturar livremente as metáforas para sugerir como tal equipe poderia funcionar se ampliada para incluir todo o suporte concebível.

O cirurgião. Mills o chama de *programador chefe*. Ele define pessoalmente as especificações funcionais e de desempenho, projeta o programa, codifica, testa e escreve sua documentação. Ele escreve em uma linguagem de programação estruturada como PL / I, e tem acesso efetivo a um sistema de computação que não apenas executa seus testes, mas também armazena as várias versões de seus programas, permite fácil atualização de arquivos e fornece edição de texto para sua documentação. Ele precisa de grande talento, dez anos de experiência e sistemas consideráveis e conhecimento de aplicativos, seja em matemática aplicada, manipulação de dados de negócios ou qualquer outra coisa.

O copiloto. Ele é o alter ego do cirurgião, capaz de fazer qualquer parte do trabalho, mas é menos experiente. Sua principal função é compartilhar o design como pensador, discutidor e avaliador. O cirurgião experimenta idéias com ele, mas não está limitado por seus conselhos. O copiloto freqüentemente representa sua equipe em discussões de função e interface com outras equipes. Ele conhece todo o código intimamente. Tenho pesquisas de estratégias alternativas de design. Ele obviamente serve como seguro contra desastres para o cirurgião. Ele pode até escrever código, mas não é responsável por nenhuma parte do código.

O administrador. O cirurgião é o chefe e deve ter a última palavra sobre pessoal, aumentos, espaço e assim por diante, mas não deve gastar quase nenhum de seu tempo com essas questões. Portanto, ele precisa de um administrador profissional que lida com dinheiro, pessoas, espaço e máquinas e que interage com a máquina administrativa do resto da organização. Baker que o administrador tem um trabalho de tempo integral apenas se o projeto sugerir que ele tem requisitos substanciais legais, contratuais, de relatórios ou financeiros devido ao relacionamento usuário-produtor. Caso contrário, um administrador pode servir a duas equipes.

O editor. O cirurgião é responsável por gerar a documentação - para maior clareza ele deve escrevê-la. Isso é verdade tanto para as descrições externas quanto para as internas. O editor, entretanto, pega o rascunho ou manuscrito ditado produzido pelo cirurgião e o critica, retrabalha, fornece referências e bibliografia, nutre-o por várias versões e supervisiona a mecânica da produção.

Duas secretárias. O administrador e o editor precisarão, cada um, de uma secretária; a secretária do administrador cuidará da correspondência do projeto e dos arquivos não relacionados ao produto.

O secretário do programa. Ele é responsável por manter todos os registros técnicos da equipe em uma biblioteca de produtos de programação. O balconista é treinado como secretário e é responsável tanto pelos arquivos legíveis por máquina quanto pelos humanos.

Todas as entradas do computador vão para o balconista, que as registra e digita, se necessário. As listagens de saída voltam para ele para serem arquivadas e indexadas. As execuções mais recentes de qualquer modelo são mantidas em um bloco de notas de status; todos os anteriores são arquivados em um arquivo cronológico.

Absolutamente vital para o conceito de Mills é a transformação da programação "de arte privada para prática pública", tornando *lá* o computador fica visível para todos os membros da equipe e identifica todos os programas e dados como propriedade da equipe, não propriedade privada.

A função especializada do secretário do programa alivia os programadores de tarefas administrativas, sistematiza e garante o desempenho adequado

3. 4 A Equipe Cirúrgica

cumprimento daquelas tarefas frequentemente negligenciadas e realça o ativo mais valioso da equipe - seu produto de trabalho. Claramente, o conceito estabelecido acima pressupõe execuções em lote. Quando terminais interativos são usados, particularmente aqueles sem saída impressa, as funções do secretário do programa não diminuem, mas mudam. Agora ele registra todas as atualizações das cópias do programa da equipe a partir de cópias de trabalho privadas, ainda lida com todas as execuções em lote e usa seu próprio recurso interativo para controlar a integridade e a disponibilidade do produto em crescimento.

O ferreiro. Os serviços de edição de arquivos, edição de texto e depuração interativa agora estão prontamente disponíveis, de forma que uma equipe raramente precisará de sua própria máquina e equipe de operação da máquina. Mas esses serviços devem estar disponíveis com resposta e confiabilidade inquestionavelmente satisfatórias; e o cirurgião deve ser o único juiz da adequação do serviço à sua disposição. Ele precisa de um ferreiro, responsável por garantir essa adequação do serviço básico e por construir, manter e atualizar ferramentas especiais - principalmente serviços de informática interativos - necessários para sua equipe. Cada equipe precisará de seu próprio ferreiro, independentemente da excelência e confiabilidade de qualquer serviço fornecido centralmente, pois seu trabalho é cuidar das ferramentas necessárias ou desejadas por *seu* cirurgião, independentemente das necessidades de qualquer outra equipe. O construtor de ferramentas freqüentemente construirá utilitários especializados, procedimentos catalogados, bibliotecas de macros.

O testador. O cirurgião precisará de um banco de casos de teste adequados para testar partes de seu trabalho à medida que o escreve e, em seguida, para testar tudo. O testador é, portanto, um adversário que concebe casos de teste de sistema a partir das especificações funcionais e um assistente que concebe dados de teste para a depuração diária. Ele também planejaria sequências de teste e configuraria a estrutura necessária para os testes de componentes.

O advogado da língua. Na época em que Algol surgiu, as pessoas começaram a reconhecer que a maioria das instalações de computador tem uma ou duas pessoas que se deliciam com o domínio das complexidades de uma linguagem de programação. E esses especialistas revelaram-se muito úteis e amplamente consultados. O talento aqui é bastante diferente daquele do cirurgião, que é principalmente um projetista de sistemas e que pensa

representações. O advogado da linguagem pode descobrir uma maneira limpa e eficiente de usar a linguagem para fazer coisas difíceis, obscuras ou complicadas. Freqüentemente, ele precisará fazer pequenos estudos (dois ou três dias) sobre uma boa técnica. Um advogado especializado em línguas pode atender a dois ou três cirurgiões.

É assim, então, que 10 pessoas podem contribuir em funções bem diferenciadas e especializadas em uma equipe de programação construída no modelo cirúrgico.

Como funciona

A equipe que acabou de definir atende aos desideratos de várias maneiras. Dez pessoas, sete delas profissionais, estão trabalhando no problema, mas o sistema é produto de uma mente - ou no máximo duas, atuando *um anime*.

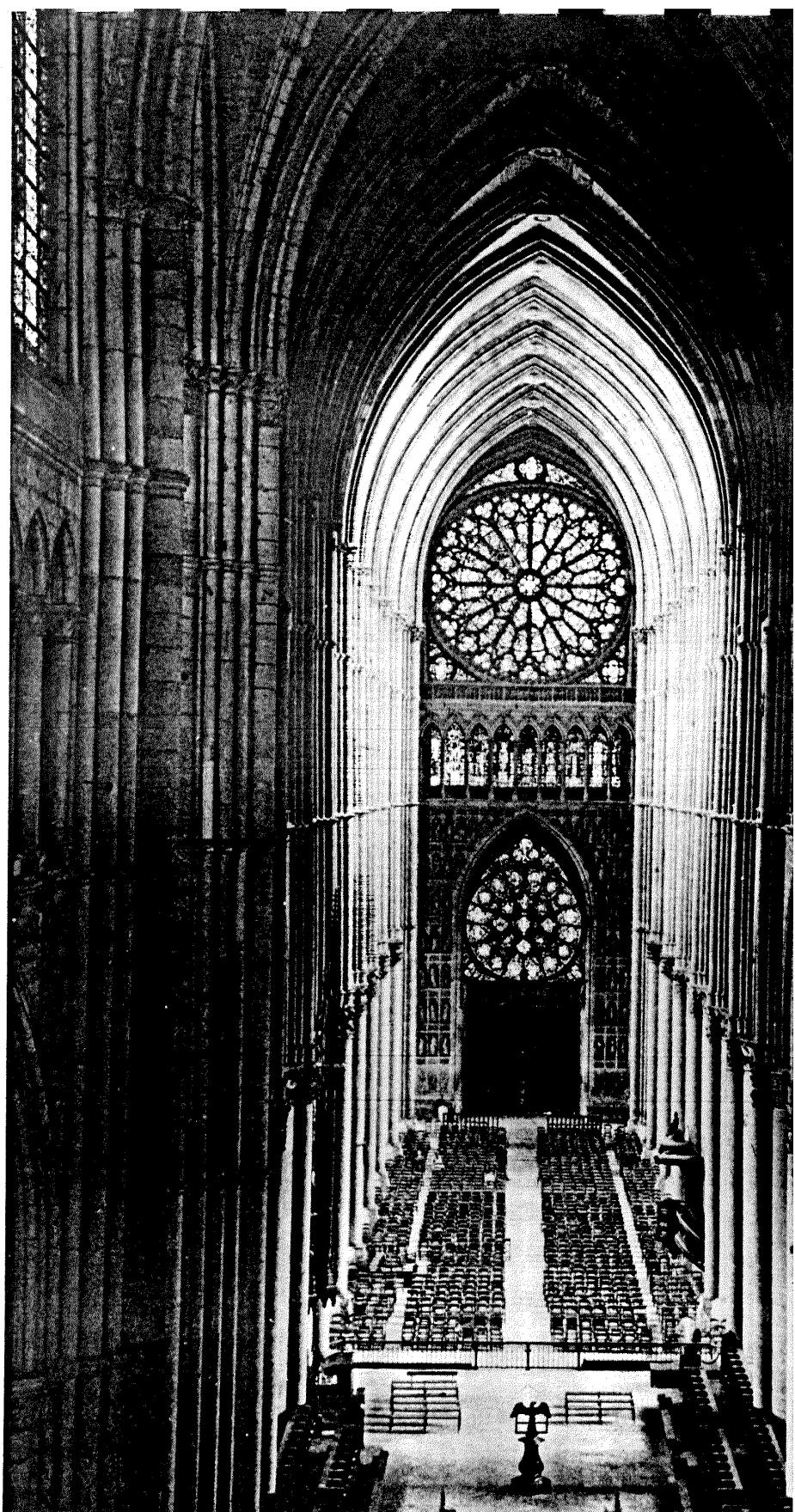
Observe, em particular, as diferenças entre uma equipe de dois programadores convencionalmente organizados e a equipe cirurgião-copiloto. Primeiro, na equipe convencional os parceiros dividem o trabalho, e cada um é responsável pelo desenho e execução de parte do trabalho. Na equipe cirúrgica, o cirurgião e o copiloto são, cada um, conhecedor de todo o design e de todo o código. Isso economiza o trabalho de alocação de espaço, acessos ao disco, etc. Também garante a integridade conceitual da obra.

Em segundo lugar, na equipe convencional os parceiros são iguais e as inevitáveis diferenças de julgamento devem ser discutidas ou comprometidas. Uma vez que o trabalho e os recursos são divididos, as diferenças de julgamento estão confinadas à estratégia geral e à interface, mas são compostas por diferenças de interesse - por exemplo, cujo espaço será usado como buffer. Na equipe cirúrgica, não há diferenças de interesse, e as diferenças de julgamento são resolvidas pelo cirurgião unilateralmente. Essas duas diferenças - a falta de divisão do problema e a relação superior-subordinado - possibilitam a atuação da equipe cirúrgica. *um anime*.

No entanto, a especialização de função do restante da equipe é a chave para sua eficiência, pois permite um padrão de comunicação radicalmente mais simples entre os membros, como mostra a Fig. 3.1.

4

Aristocracia, democracia e design de sistema



4

Aristocracia, Democracia, e design de sistema

Esta grande igreja é uma obra de arte incomparável. Não há aridez nem confusão nos princípios que ele apresenta. . ,

É o apogeu de um estilo, a obra de artistas que compreenderam e assimilaram todos os sucessos de seus predecessores, em plena posse das técnicas de sua época, mas utilizando-as sem exibições indiscretas nem proezas gratuitas de habilidade.

Foi Jean d'Orbais quem sem dúvida concebeu a planta geral do edifício, uma planta que foi respeitada, pelo menos nos seus elementos essenciais, pelos seus sucessores. Esta é uma das razões da extrema coerência e unidade do edifício.

REIMS CATEDRAL GUIA,

Fotografias Emmanuel Boudot-Lamotte

Integridade Conceitual

A maioria das catedrais europeias mostra diferenças no plano ou estilo arquitetônico entre as peças construídas em gerações diferentes por construtores diferentes. Os construtores posteriores foram tentados a "aprimorar" os designs dos anteriores, para refletir tanto as mudanças na moda quanto as diferenças no gosto individual. Assim, o pacífico transepto normando confina e contradiz a alta nave gótica, e o resultado proclama o orgulho dos construtores tanto quanto a glória de Deus.

Contra eles, a unidade arquitetônica de Reims está em glorioso contraste. A alegria que mexe com o observador vem tanto da integridade do design quanto de quaisquer excelências em particular. Como diz o guia, essa integridade foi alcançada pela abnegação de oito gerações de construtores, cada um dos quais sacrificou algumas de suas idéias para que o todo pudesse ser de puro design. O resultado proclama não apenas a glória de Deus, mas também Seu poder para salvar os homens caídos de seu orgulho.

Embora não tenham levado séculos para serem construídos, a maioria dos sistemas de programação reflete uma desunião conceitual muito pior do que a das catedrais. Normalmente, isso surge não de uma sucessão em série de designers mestres, mas da separação do design em muitas tarefas realizadas por muitos homens.

Vou argumentar que a integridade conceitual é a consideração mais importante no projeto do sistema. É melhor ter um sistema omitindo certos recursos e melhorias anômalas, mas para refletir um conjunto de idéias de design, do que ter um que contenha muitas idéias boas, mas independentes e descoordenadas. Neste capítulo e nos próximos dois, examinaremos as consequências deste tema para o design de sistemas de programação:

- Como a integridade conceitual deve ser alcançada?
- Este argumento não implica uma elite, ou aristocracia de arquitetos, e uma horda de implementadores plebeus cujos talentos criativos e ideias são suprimidos?

- Como evitar que os arquitetos mergulhem no azul com especificações não implementáveis ou caras?
- Como garantir que todos os detalhes insignificantes de uma especificação arquitetônica sejam comunicados ao implementador, devidamente compreendidos por ele e incorporados com precisão ao produto?

Atingindo a Integridade Conceitual

O objetivo de um sistema de programação é tornar um computador fácil de usar. Para fazer isso, ele fornece linguagens e vários recursos que são, na verdade, programas invocados e controlados por recursos de linguagem. Mas esses recursos são comprados por um preço: a descrição externa de um sistema de programação é dez a vinte vezes maior que a descrição externa do próprio sistema de computador. O usuário acha muito mais fácil especificar qualquer função em particular, mas há muito mais opções para escolher e muito mais opções e formatos para lembrar.

A facilidade de uso é aprimorada apenas se o tempo ganho na especificação funcional exceder o tempo perdido no aprendizado, lembrança e pesquisa de manuais. Com os sistemas de programação modernos, esse ganho excede o custo, mas nos últimos anos a relação entre o ganho e o custo parece ter caído à medida que funções cada vez mais complexas foram adicionadas. Estou assombrado pela memória da facilidade de uso do IBM 650, mesmo sem um montador ou qualquer outro software.

Como a facilidade de uso é o objetivo, essa relação entre a função e a complexidade conceitual é o teste final do projeto do sistema. Nem a função por si só, nem a simplicidade por si definem um bom design.

Este ponto é amplamente mal compreendido. O Operating System / 360 é saudado por seus construtores como o melhor já construído, porque indiscutivelmente tem mais funções. Função, e não simplicidade, sempre foi a medida de excelência para seus designers. Por outro lado, o Sistema de Compartilhamento de Tempo para o PDP-10 é saudado por seus construtores como o melhor, devido à sua simplicidade e à escassez

de seus conceitos. Em qualquer medida, no entanto, sua função não é nem mesmo na mesma classe que a do OS / 360. Assim que a facilidade de uso é considerada o critério, cada um deles é visto como desequilibrado, alcançando apenas metade do objetivo verdadeiro.

Para um determinado nível de função, entretanto, esse sistema é o melhor em que se possa especificar as coisas com a maior simplicidade e objetividade. *Simplicidade* não é o suficiente. A linguagem TRAC de Mooers e o Algol 68 alcançam simplicidade medida pelo número de conceitos elementares distintos. Eles não são, no entanto, *para a frente*. A expressão das coisas que se deseja fazer muitas vezes requer combinações complexas e inesperadas das instalações básicas. Não é suficiente aprender os elementos e regras de combinação; deve-se também aprender o uso idiomático, toda uma tradição de como os elementos são combinados na prática. Simplicidade e franqueza procedem da integridade conceitual. Cada parte deve refletir as mesmas filosofias e o mesmo equilíbrio de desiderata. Cada parte deve até usar as mesmas técnicas de sintaxe e noções análogas de semântica. A facilidade de uso, então, dita a unidade de design, a integridade conceitual.

Aristocracia e Democracia

A integridade conceitual, por sua vez, determina que o projeto deve proceder de uma mente ou de um número muito pequeno de mentes ressonantes concordantes.

As pressões de programação, no entanto, ditam que a construção do sistema precisa de muitas mãos. Duas técnicas estão disponíveis para resolver esse dilema. O primeiro é uma divisão cuidadosa de trabalho entre arquitetura e implementação. A segunda é a nova maneira de estruturar equipes de implementação de programação discutida no capítulo anterior.

A separação do esforço arquitetônico da implementação é uma maneira muito poderosa de obter integridade conceitual em projetos muito grandes. Eu mesmo já vi isso ser usado com grande sucesso no computador Stretch da IBM e na linha de produtos de computador System / 360.

Eu vi falha por falta de aplicativo no Operating System / 360.

Pelo *arquitetura* de um sistema, refiro-me à especificação completa e detalhada da interface do usuário. Para um computador, este é o manual de programação. Para um compilador, é o manual da linguagem. Para um programa de controle, são os manuais do idioma ou idiomas usados para invocar suas funções. Para todo o sistema é a união dos manuais que o usuário deve consultar para realizar todo o seu trabalho.

O arquiteto de um sistema, como o arquiteto de um edifício, é o agente do usuário. É seu trabalho trazer conhecimento profissional e técnico para apoiar o interesse puro do usuário, em oposição aos interesses do vendedor, do fabricante, etc.⁸

A arquitetura deve ser cuidadosamente diferenciada da implementação. Como disse Blaauw, "Onde a arquitetura diz *That* acontece, a implementação diz *Como as* é feito para acontecer."³ Ele dá como um exemplo simples um relógio, cuja arquitetura consiste no mostrador, nos ponteiros e no botão de corda. Quando uma criança aprende essa arquitetura, ela pode contar as horas com a mesma facilidade de um relógio de pulso ou de uma torre de igreja. A implementação, no entanto, e sua realização, descreve o que acontece dentro do gabinete - acionado por qualquer um dos muitos mecanismos e controle de precisão por qualquer um de muitos.

No System / 360, por exemplo, uma única arquitetura de computador é implementada de maneira bastante diferente em cada um dos nove modelos. Por outro lado, uma única implementação, o fluxo de dados, memória e microcódigo do Modelo 30, serve em momentos diferentes para quatro arquiteturas diferentes: um computador System / 360, um canal multiplex com até 224 subcanais logicamente independentes, um canal seletor e um computador 1401.⁴

A mesma distinção é igualmente aplicável a sistemas de programação. Existe um Fortran IV padrão dos EUA. Esta é a arquitetura de muitos compiladores. Dentro desta arquitetura, muitas implementações são possíveis: texto no núcleo ou compilador no núcleo, compilação rápida ou otimização, direcionado à sintaxe ou *Ad hoc*. Da mesma forma, qualquer linguagem assembler ou linguagem de controle de trabalho admite muitas implementações do montador ou escalonador.

46 Aristocracy, Democracy e SystemDesign

Agora podemos lidar com a questão profundamente emocional de aristocracia versus democracia. Não são os arquitetos uma nova aristocracia, uma elite intelectual, criada para dizer aos pobres implementadores idiotas o que fazer? Todo o trabalho criativo não foi sequestrado para essa elite, deixando os implementadores como engrenagens da máquina? Não se obterá um produto melhor obtendo as boas ideias de toda a equipe, seguindo uma filosofia democrática, em vez de restringir o desenvolvimento de especificações a poucas?

Quanto à última pergunta, é a mais fácil. Certamente não vou afirmar que apenas os arquitetos terão boas idéias arquitetônicas. Freqüentemente, o conceito novo vem de um implemento ou de um usuário. No entanto, toda minha própria experiência me convence, e tentei mostrar, que a integridade conceitual de um sistema determina sua facilidade de uso. Bons recursos e ideias que não se integram aos conceitos básicos de um sistema devem ser deixados de fora. Se aparecerem muitas dessas idéias importantes, mas incompatíveis, descartamos todo o sistema e começamos novamente em um sistema integrado com diferentes conceitos básicos.

Quanto à acusação de aristocracia, a resposta deve ser sim e não. Sim, no sentido de que deve haver poucos arquitetos, seu produto deve durar mais tempo do que o de um implementador, e o arquiteto fica no foco de forças que ele deve resolver no interesse do usuário. Se um sistema deve ter integridade conceitual, alguém deve controlar os conceitos. Essa é uma aristocracia que não precisa de desculpas.

Não, porque a definição de especificações externas não é um trabalho mais criativo do que o projeto de implementações. É apenas um trabalho criativo diferente. O design de uma implementação, dada uma arquitetura, requer e permite tanta criatividade de design, tantas novas ideias e tanto brilho técnico quanto o design das especificações externas. Na verdade, a relação custo-desempenho do produto dependerá mais fortemente do implementador, assim como a facilidade de uso depende mais do arquiteto.

Existem muitos exemplos de outras artes e ofícios que levam a crer que a disciplina é boa para a arte. Na verdade, o de um artista

o aforismo afirma: "A forma é libertadora." Os piores edifícios são aqueles cujo orçamento era grande demais para os propósitos a serem atendidos. A produção criativa de Bach dificilmente parece ter sido reprimida pela necessidade de produzir uma pantata de forma limitada a cada semana. *eu* certeza de que o computador Stretch teria uma arquitetura melhor se fosse mais restrita; as restrições impostas pelo orçamento do System / 360 Model 30 foram, em minha opinião, inteiramente benéficas para a arquitetura do Model 75.

Da mesma forma, observo que o fornecimento externo de uma arquitetura aumenta, e não restringe, o estilo criativo de um grupo de implementação. Eles se concentram imediatamente na parte do problema que ninguém abordou, e as invenções começam a fluir. Em um grupo de implementação irrestrito, a maior parte do pensamento e do debate vai para as decisões arquitetônicas, e a implementação adequada recebe pouca atenção.⁵

Este efeito, que tenho visto muitas vezes, é confirmado por RW Conway, cujo grupo em Cornell construiu o compilador PL / C para a linguagem PL / I. Ele diz: "Finalmente decidimos implementar a linguagem inalterada e sem melhorias, pois os debates sobre a linguagem teriam exigido todo o nosso esforço."⁶

O que o implementador faz durante a espera?

É uma experiência muito humilhante cometer um erro multimilionário, mas também é muito memorável. Lembro-me vividamente da noite em que decidimos como organizar a redação real das especificações externas para OS / 360. O gerente de arquitetura, o gerente de implementação do programa de controle e eu estávamos debatendo o plano, o cronograma e a divisão de responsabilidades.

O gerente de arquitetura tinha 10 bons homens. Afirmei que eles poderiam escrever as especificações e fazê-lo direito. Levaria dez meses, três a mais do que a programação permitia.

O gerente do programa de controle tinha 150 homens. Afirmei que eles poderiam preparar as especificações, com a coordenação da equipe de arquitetura; seria bem feito e prático, e ele poderia fazê-lo dentro do prazo. Além disso, se a equipe de arquitetura fizesse isso, seus 150 homens ficariam sentados girando os polegares por dez meses.

48 Aristocracy, Democracy e SystemDesign

A isso, o gerente de arquitetura respondeu que se eu desse a responsabilidade à equipe do programa de controle, o resultado seria *não* na verdade, chegaria a tempo, mas também atrasaria três meses e seria de qualidade muito inferior. Eu fiz, e foi. Ele estava certo em ambos os casos. Além disso, a falta de integridade conceitual tornou o sistema muito mais caro para construir e alterar, e eu estimaria que acrescentou um ano ao tempo de depuração.

Muitos fatores, é claro, influenciaram essa decisão equivocada; mas o opressor era o tempo programado e o apelo de colocar todos aqueles 150 implementadores para trabalhar. É esse canto de sereia cujos perigos mortais eu agora tornaria visíveis.

Quando é proposto que uma pequena equipe de arquitetura de fato escreva todas as especificações externas para um computador ou sistema de programação, os implementadores levantam três objeções:

- As especificações serão muito ricas em funções e não refletirão considerações práticas de custo.
- Os arquitetos obterão toda a diversão criativa e excluirão a inventividade dos implementadores.

«Os muitos implementadores terão que ficar de braços cruzados enquanto o as especificações vêm por meio de um funil estreito que é a equipe de arquitetura.

O primeiro deles é um perigo real e será tratado no próximo capítulo. Os outros dois são ilusões, pura e simples. Como vimos acima, a implementação também é uma atividade criativa de primeira ordem. A oportunidade de ser criativo e inventivo na implementação não é significativamente diminuída pelo trabalho dentro de uma determinada especificação externa, e a ordem da criatividade pode até ser aprimorada por essa disciplina. O produto total certamente será.

A última objeção é de tempo e fases. Uma resposta rápida é abster-se de contratar implementadores até que as especificações estejam completas. Isso é o que é feito quando um edifício é construído.

No negócio de sistemas de computador, no entanto, o ritmo é mais rápido e deseja-se reduzir a programação o máximo possível. Até que ponto a especificação e a construção podem ser sobrepostas?

Como Blaauw aponta, o esforço criativo total envolve três fases distintas: arquitetura, implementação e realização. Acontece que eles podem, de fato, ser iniciados em paralelo e prosseguir simultaneamente.

No projeto de computador, por exemplo, o implementador pode começar assim que tiver suposições relativamente vagas sobre o manual, ideias um pouco mais claras sobre a tecnologia e objetivos de custo e desempenho bem definidos. Ele pode começar a projetar fluxos de dados, sequências de controle, conceitos de embalagem bruta e assim por diante. Ele concebe ou adapta as ferramentas de que precisará, especialmente o sistema de manutenção de registros, incluindo o sistema de automação de projeto.

Enquanto isso, no nível de realização, circuitos, placas, cabos, quadros, fontes de alimentação e memórias devem ser projetados, refinados e documentados. Este trabalho prossegue em paralelo com a arquitetura e implementação.

A mesma coisa vale para o projeto do sistema de programação. Muito antes de as especificações externas estarem completas, o implemento tem muito o que fazer. Dadas algumas aproximações grosseiras quanto à função do sistema que será finalmente incorporado nas especificações externas, ele pode prosseguir. Ele deve ter objetivos de espaço e tempo bem definidos. Ele deve saber a configuração do sistema em que seu produto deve ser executado. Em seguida, ele pode começar a projetar limites de módulo, estruturas de tabela, quebras de passagem ou fase, algoritmos e todos os tipos de ferramentas. Algum tempo também deve ser gasto na comunicação com o arquiteto.

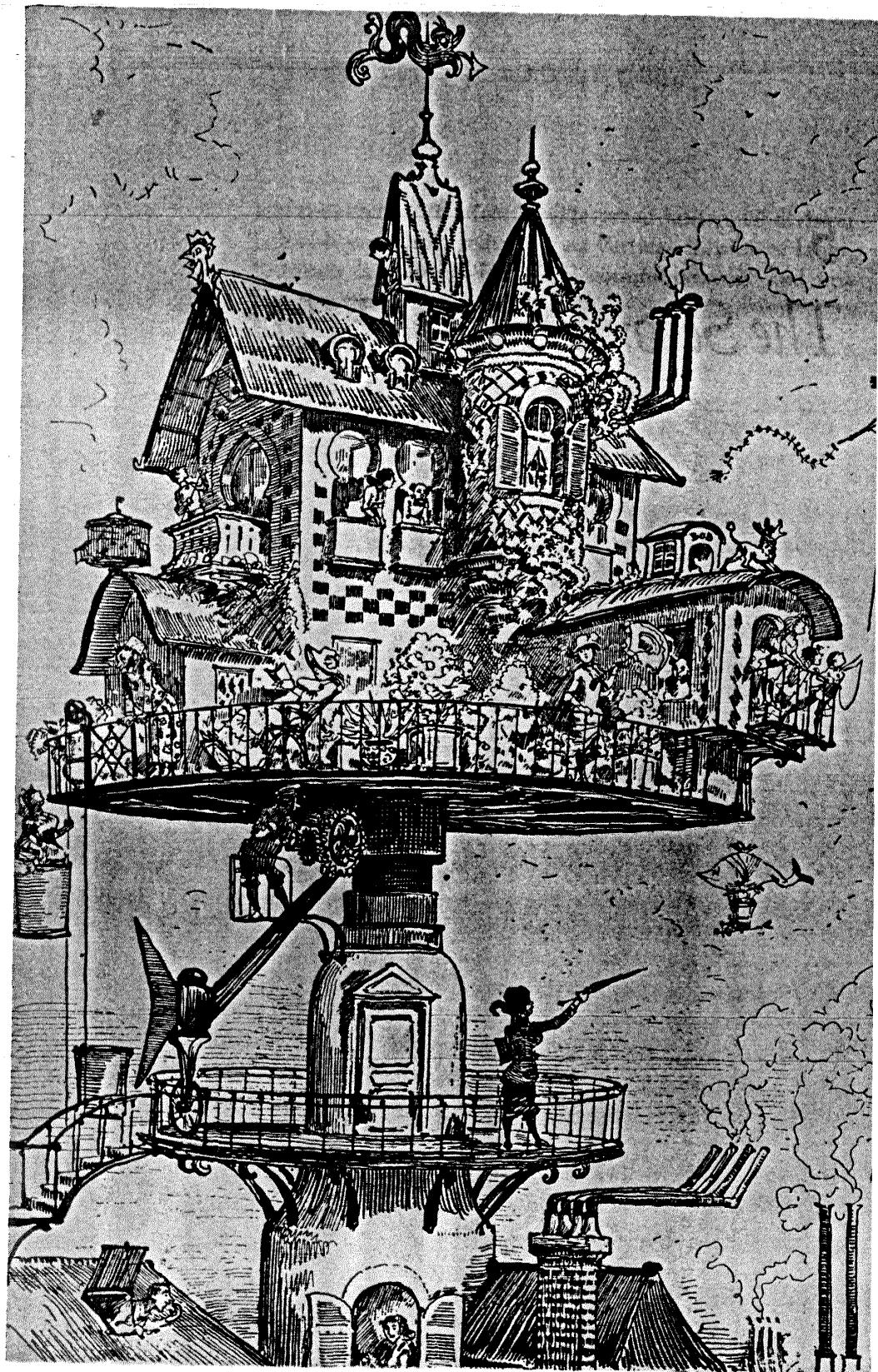
Enquanto isso, no nível de realização, também há muito a ser feito. A programação também tem uma tecnologia. Se a máquina for nova, muito trabalho deve ser feito em convenções de sub-rotinas, técnicas de supervisão, busca e algoritmos de classificação.⁷

A integridade conceitual exige que um sistema reflita uma única filosofia e que a especificação vista pelo usuário flua em alguns minutos. Devido à divisão real do trabalho em arquitetura, implementação e realização, no entanto, isso não significa que um sistema assim projetado levará mais tempo para ser construído. A experiência mostra o oposto, que o sistema integral se une mais rápido e

leva menos tempo para testar. Com efeito, uma divisão horizontal generalizada do trabalho foi drasticamente reduzida por uma divisão vertical do trabalho, e o resultado são comunicações radicalmente simplificadas e integridade conceitual aprimorada.

5

O efeito do segundo sistema



5

O efeito do segundo sistema

Adde parvum parvo magnus acervus erit.

{Adicione pouco a pouco e haverá uma pilha de Ug.]

OVID

Casa virada para o tráfego aéreo. Litografia, Paris, 1882 de
Le Vingtiiime Stecle por A. Robida

Se separarmos a responsabilidade pela especificação funcional da responsabilidade pela construção de um produto rápido e barato, que disciplina limita o entusiasmo inventivo do arquiteto?

A resposta fundamental é uma comunicação completa, cuidadosa e compreensiva entre arquiteto e construtor. No entanto, existem respostas mais refinadas que merecem atenção.

Disciplina interativa para o arquiteto

O arquiteto de um edifício trabalha contra um orçamento, usando técnicas de estimativa que são posteriormente confirmadas ou corrigidas pelas propostas dos empreiteiros. Muitas vezes acontece que todos os lances excedem o orçamento. O arquiteto então revisa sua técnica de estimativa para cima e seu projeto para baixo para outra iteração. Ele talvez possa sugerir aos empreiteiros maneiras de implementar seu projeto de forma mais barata do que eles haviam planejado.

Um processo análogo governa o arquiteto de um sistema de computador ou sistema de programação. Ele tem, no entanto, a vantagem de receber propostas do empreiteiro em muitos pontos iniciais de seu projeto, quase sempre que ele as solicitar. Ele geralmente tem a desvantagem de trabalhar com apenas um empreiteiro, que pode aumentar ou diminuir suas estimativas para refletir seu prazer com o projeto. Na prática, a comunicação inicial e contínua pode dar ao arquiteto boas leituras de custo e ao construtor confiança no projeto, sem obscurecer a divisão clara de responsabilidades.

O arquiteto tem duas respostas possíveis quando confrontado com uma estimativa que é muito alta: corte o projeto ou desafie a estimativa sugerindo implementações mais baratas. Este último é inherentemente uma atividade geradora de emoção. O arquiteto agora está desafiando a maneira do construtor de fazer o trabalho do construtor. Para ter sucesso, o arquiteto deve

- lembre-se de que o construtor tem a responsabilidade inventiva e criativa pela implementação; assim o arquiteto sugere, não dita;

- esteja sempre preparado para sugerir *para* forma de implementar qualquer coisa que ele especificar, e esteja preparado para aceitar qualquer outra forma que atenda aos objetivos também;
- lidar de forma discreta e privada com essas sugestões;
- esteja pronto para abrir mão do crédito por sugestões de melhorias.

Normalmente, o construtor irá contra-atacar sugerindo mudanças na arquitetura. Freqüentemente, ele está certo - alguns recursos menores podem ter custos inesperadamente altos quando a implementação é resolvida.

Autodisciplina - O efeito do segundo sistema

O primeiro trabalho de um arquiteto tende a ser simples e limpo. Ele sabe que não sabe o que está fazendo, então o faz com cuidado e com grande moderação.

Enquanto ele desenha o primeiro trabalho, folho após folho e enfeite após enfeite ocorrem para ele. Estes são armazenados para serem usados na "próxima vez". Mais cedo ou mais tarde, o primeiro sistema estará concluído, e o arquiteto, com firmeza, confiança e um domínio demonstrado dessa classe de sistemas, está pronto para construir um segundo sistema.

Este segundo é o sistema mais perigoso que um homem já projetou. Quando ele fizer o terceiro e os posteriores, suas experiências anteriores se confirmarão quanto às características gerais de tais sistemas, e suas diferenças identificarão as partes de sua experiência que são particulares e não generalizáveis.

A tendência geral é projetar demais o segundo sistema, usando todas as idéias e enfeites que foram cuidadosamente desviados no primeiro. O resultado, como diz Ovid, é uma "grande pilha". Por exemplo, considere a arquitetura IBM 709, posteriormente incorporada ao 7090. Esta é uma atualização, um segundo sistema para o 704 muito bem-sucedido e limpo. O conjunto de operações é tão rico e abundante que apenas cerca de metade dele era usado regularmente.

Considere como um caso mais forte a arquitetura, implementação e até mesmo a realização do computador Stretch, uma saída para o

desejos inventivos reprimidos de muitas pessoas, e um segundo sistema para a maioria delas. Como Strachey diz em uma revisão:

*Tenho a impressão de que Stretch é, de alguma forma, o fim de uma linha de desenvolvimento. Como alguns dos primeiros programas de computador, é imensamente engenhoso, complicado e eficaz, mas ao mesmo tempo rude, esbanjador e deselegante, e sentimos que deve haver uma maneira melhor de fazer as coisas. **

Operating System / 360 foi o segundo sistema para a maioria de seus designers. Grupos de seus designers vieram da construção do sistema operacional de disco 14107010, do sistema operacional Stretch, do sistema de tempo real Project Mercury e do IBSYS para o 7090. Quase ninguém tinha experiência com *dois* sistemas operacionais anteriores.⁸ Portanto, OS / 360 é um excelente exemplo do efeito do segundo sistema, um trecho da arte do software ao qual tanto as recomendações quanto as reprovações da crítica de Strachey se aplicam inalteradas.

Por exemplo, o OS / 360 dedica 26 bytes da rotina de mudança de data residente permanentemente para o tratamento adequado de 31 de dezembro em anos bissextos (quando é o Dia 366). Isso pode ter sido deixado para a operadora.

O efeito do segundo sistema tem outra manifestação um tanto diferente do puro embelezamento funcional. Essa é uma tendência de refinar técnicas cuja própria existência se tornou obsoleta por mudanças nas suposições do sistema básico. OS / 360 tem muitos exemplos disso.

Considere o editor de ligação, projetado para carregar programas compilados separadamente e resolver suas referências cruzadas. Além dessa função básica, ele também lida com sobreposições de programa. É uma das melhores instalações de sobreposição já construídas. Ele permite que a estruturação da sobreposição seja feita externamente, no momento da ligação, sem ser projetada no código-fonte. Ele permite que a estrutura de sobreposição seja alterada de execução para execução sem recompilação. Ele fornece uma rica variedade de opções e instalações úteis. Em certo sentido, é o culminar de anos de desenvolvimento da técnica de sobreposição estática.

No entanto, é também o último e melhor dos dinossauros, pois pertence a um sistema em que a multiprogramação é o modo normal e a alocação dinâmica do núcleo a suposição básica. Isso está em conflito direto com a noção de usar sobreposições estáticas. O sistema funcionaria muito melhor se os esforços dedicados ao gerenciamento da sobreposição tivessem sido gastos para tornar a alocação dinâmica do núcleo e os recursos de referência cruzada dinâmica realmente rápidos!

Além disso, o editor de vinculação requer muito espaço e contém muitas sobreposições que, mesmo quando usado apenas para vinculação sem gerenciamento de sobreposição, é mais lento do que a maioria dos compiladores do sistema. A ironia disso é que o objetivo do vinculador é evitar a recompilação. Como um patinador cujo estômago fica à frente de seus pés, o refinamento prosseguiu até que as suposições do sistema tivessem sido superadas.

O recurso de depuração TESTRAN é outro exemplo dessa tendência. É o culminar das instalações de depuração em lote, fornecendo recursos de snapshot e dump de memória verdadeiramente elegantes. Ele usa o conceito de seção de controle e uma técnica de gerador engenhosa para permitir rastreamento seletivo e captura de imagem sem sobrecarga interpretativa ou recompilação. Os conceitos criativos do Sistema Operacional Compartilhado 3 pois o 709 foi trazido à floração completa.

Enquanto isso, toda a noção de depuração em lote sem recompilação estava se tornando obsoleta. Os sistemas de computação interativos, usando intérpretes de linguagem ou compiladores incrementais, forneceram o desafio mais fundamental. Mas mesmo em sistemas em lote, o surgimento de compiladores de compilação rápida / execução lenta tornou a depuração de nível de origem e captura de imagem a técnica preferida. Quão melhor o sistema teria sido se o esforço do TESTRAN tivesse sido dedicado a construir os recursos interativos e de compilação rápida mais cedo e melhor!

Ainda outro exemplo é o planejador, que fornece recursos realmente excelentes para gerenciar um fluxo de trabalho de lote fixo. Em um sentido real, este agendador é o segundo sistema refinado, aprimorado e embelezado que sucede ao Sistema Operacional de Disco 1410-7010,

§8 O efeito do segundo sistema

um sistema em lote não multiprogramado, exceto para entrada-saída e destinado principalmente para aplicativos de negócios. Como tal, o agendador OS / 360 é bom. Mas é quase totalmente não influenciado pelas necessidades do OS / 360 de entrada de trabalho remoto, multiprogramação e subsistemas interativos permanentemente residentes. Na verdade, o design do escalonador torna isso difícil.

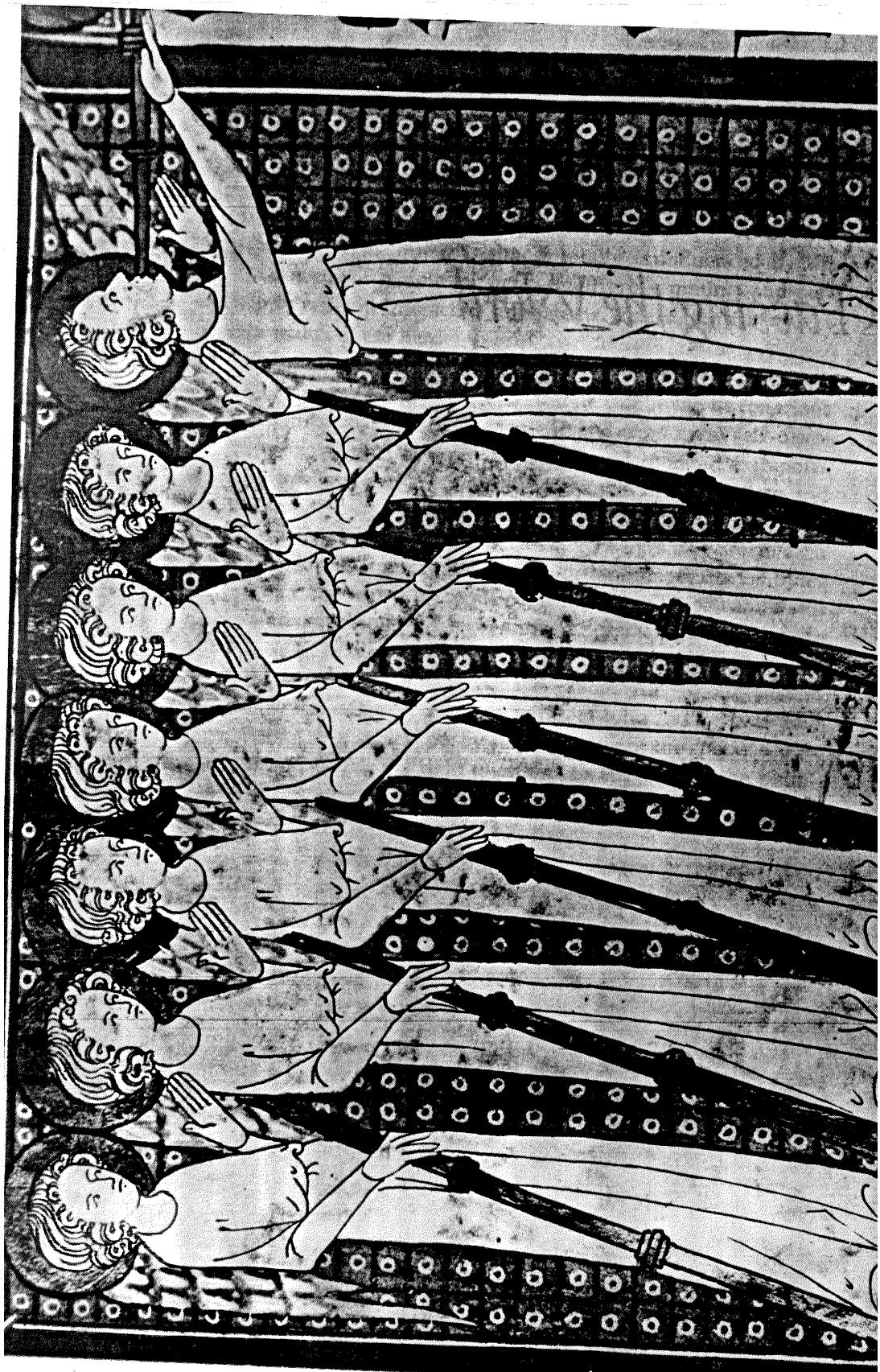
Como o arquiteto evita o efeito do segundo sistema? Bem, obviamente ele não pode ignorar seu segundo sistema. Mas ele pode estar ciente dos perigos peculiares desse sistema e exercer autodisciplina extra para evitar a ornamentação funcional e para evitar a extrapolação de funções que são evitadas por mudanças nas suposições e propósitos.

Uma disciplina que abrirá os olhos de um arquiteto é atribuir a cada pequena função um valor: capacidade x não vale mais do que m bytes de memória e n microsegundos por invocação. Esses valores guiarão as decisões iniciais e servirão durante a implementação como um guia e um aviso para todos.

Como o gerente de projeto evita o efeito do segundo sistema? Ao insistir em um arquiteto sênior que tenha pelo menos dois sistemas sob sua responsabilidade. Além disso, ao ficar atento às tentações especiais, ele pode fazer as perguntas certas para garantir que os conceitos e objetivos filosóficos sejam totalmente refletidos no projeto detalhado.

6

Passando a Palavra



6

Passando a Palavra

Ele vai sentar aqui e dizer: "Faça isso! Faça aquilo!" E nada vai acontecer.

HARRYS. TRUMAN. NO PODER PRESIDENCIAL,

"As Sete Trombetas" de *O Apocalipse Wells*, Arquivo Bettman
do século 14

Supondo que ele tenha arquitetos disciplinados e experientes e que haja muitos implementadores, como o gerente deve garantir que todos ouçam, entendam e implementem as decisões dos arquitetos? Como pode um grupo de 10 arquitetos manter a integridade conceitual de um sistema que 1.000 homens estão construindo? Toda uma tecnologia para fazer isso foi desenvolvida para o esforço de design de hardware do System / 360, e é igualmente aplicável a projetos de software.

Especificações Escritas - o Manual

O manual, ou especificação escrita, é uma ferramenta necessária, embora não suficiente. O manual é o *externo* especificação do produto. Ele descreve e prescreve todos os detalhes do que o usuário vê. Como tal, é o principal produto do arquiteto.

O ciclo de preparação ocorre continuamente, à medida que o feedback dos usuários e implementadores mostra onde o design é difícil de usar ou construir. Para o bem dos implementadores, é importante que as mudanças sejam quantizadas - que haja versões datadas aparecendo em uma programação.

O manual não deve apenas descrever tudo o que o usuário vê, incluindo todas as interfaces; também deve abster-se de descrever o que o usuário não vê. Esse é o negócio do implementador, e aí sua liberdade de design deve ser irrestrita. O arquiteto deve estar sempre preparado para mostrar *um* implementação para qualquer recurso que ele descreve, mas ele não deve tentar ditar a implementação.

O estilo deve ser preciso, completo e detalhado com precisão. Freqüentemente, um usuário se refere a uma única definição, de modo que cada uma deve repetir todos os fundamentos e, ainda assim, todos devem concordar. Isso tende a tornar a leitura dos manuais enfadonha, mas a precisão é mais importante do que a vivacidade.

A unidade do System / 360's *Princípios de Operação* deriva do fato de que apenas dois pensamentos o escreveram: o de Gerry Blaauw e o de Andris Padegs. As ideias são de cerca de dez homens, mas o lançamento dessas decisões nas especificações da prosa deve ser feito por apenas um

ou dois, se a consistência da prosa e do produto deve ser mantida. Pois a redação de uma definição exigirá uma série de minidecisões que não são de importância para um debate completo. Um exemplo no System / 360 é o detalhe de como o código de condição é definido após cada operação. *Não* trivial, no entanto, é o princípio de que tais minidecisões sejam feitas de forma consistente durante todo o processo.

Acho que a melhor escrita manual que já vi é Apêndice de Blaauw *System / 360 Principles of Operation*. Thisde-
escribas com cuidado e precisão o *limites* de compatibilidade System / 360. Ele define a compatibilidade, prescreve o que deve ser alcançado e enumera as áreas de aparência externa onde a arquitetura é intencionalmente silenciosa e onde os resultados de um modelo podem ser diferentes dos de outro, onde uma cópia de um determinado modelo pode ser diferente de outra cópia, ou onde uma cópia pode ser diferente de si mesma após uma alteração de engenharia. Este é o nível de precisão ao qual os escritores manuais aspiram, e eles devem definir o que é *não* prescrito tão cuidadosamente quanto o que é.

Definições Formais

O inglês, ou qualquer outra língua humana, não é naturalmente um instrumento de precisão para tais definições. Portanto, o redator do manual deve esforçar-se e a sua linguagem para obter a precisão necessária. Uma alternativa atraente é usar uma notação formal para tais definições. Afinal, a precisão é o principal produto do comércio, o *razão de ser* de notações formais.

Vamos examinar os méritos e fraquezas das definições formais. Conforme observado, as definições formais são precisas. Eles tendem a ser completos; as lacunas aparecem de forma mais visível, portanto, são preenchidas logo. O que falta é compreensibilidade. Com a prosa inglesa, pode-se mostrar princípios estruturais, delinear a estrutura em estágios ou níveis e dar exemplos. É possível marcar exceções prontamente e enfatizar contrastes. Mais importante, pode-se explicar *porque*. As definições formais apresentadas até agora inspiraram admiração por sua elegância e confiança em sua precisão. Mas eles exigiram

explicações em prosa para tornar seu conteúdo fácil de aprender e ensinar. Por essas razões, acho que veremos as especificações futuras consistindo em uma definição formal e uma definição de prosa.

Um antigo ditado adverte: "Nunca vá para o mar com dois cronômetros; pegue um ou três." A mesma coisa se aplica claramente à prosa e às definições formais. Se um tiver ambos, um deve ser o padrão e o outro deve ser uma descrição derivada, claramente rotulada como tal. Qualquer um pode ser o padrão principal. Algol 68 tem uma definição formal como padrão e uma definição em prosa como descritiva. PL / I tem a prosa como padrão e a descrição formal como derivada. O System / 360 também tem a prosa como padrão com uma descrição formal derivada.

Muitas ferramentas estão disponíveis para definição formal. O BackusNaur Form é familiar para definição de linguagem e é amplamente discutido na literatura.² A descrição formal de PL / I usa novas noções de sintaxe abstrata e está adequadamente descrita.³

O APL de Iverson tem sido usado para descrever máquinas, mais notavelmente o IBM 7090⁴ e System / 360.⁵

Bell e Newell propuseram novas notações para descrever configurações e arquiteturas de máquina, e as ilustraram com várias máquinas, incluindo o DECPDP-8,⁶

o 7090,⁶ e System / 360.⁷

Quase todas as definições formais acabam por incorporar ou descrever uma implementação do sistema de hardware ou software cujas partes externas estão prescrevendo. A sintaxe pode ser descrita sem isso, mas a semântica geralmente é definida fornecendo um programa que executa a operação definida. É claro que isso é uma implementação e, como tal, prescreve excessivamente a arquitetura. Portanto, deve-se ter o cuidado de indicar que a definição formal se aplica apenas a elementos externos e deve-se dizer quais são.

Não apenas uma definição formal é uma implementação, uma implementação pode servir como uma definição formal. Quando os primeiros computadores compatíveis foram construídos, essa era exatamente a técnica usada. A nova máquina deveria corresponder a uma máquina existente. O manual foi vago em alguns pontos? "Pergunte à máquina!" Um programa de teste

seria planejado para determinar o comportamento e a nova máquina seria construída para corresponder.

Um simulador programado de um sistema de hardware ou software pode funcionar exatamente da mesma maneira. É uma implementação; ele corre.

Portanto, todas as questões de definição podem ser resolvidas testando-o.

Usar uma implementação como definição tem algumas vantagens.

Todas as questões podem ser resolvidas de forma inequívoca por meio de experimentos. O debate nunca é necessário, então as respostas são rápidas.

As respostas são sempre tão precisas quanto se deseja e estão sempre corretas, por definição. Oposto a esses tem um conjunto formidável de desvantagens. A implementação pode prescrever em excesso até mesmo os externos. A sintaxe inválida sempre produz algum resultado; em um sistema policiado que resulta em uma indicação de invalidez *e nada mais*. Em um sistema não policiado, todos os tipos de efeitos colaterais podem aparecer, e eles podem ter sido usados por programadores. Quando nos comprometemos a emular o IBM 1401 no System / 360, por exemplo, ele descobriu que havia 30 "curiosidades" diferentes - efeitos colaterais de operações supostamente inválidas - que passaram a ser amplamente utilizadas e tiveram que ser consideradas como parte da definição. A implementação como definição exagerada; não dizia apenas o que a máquina deveria fazer, mas também dizia muito sobre como deveria fazê-lo.

Então, também, a implementação às vezes dará respostas inesperadas e não planejadas quando perguntas afiadas são feitas, e o *de fato* a definição freqüentemente será considerada deselegante nesses detalhes precisamente porque eles nunca receberam qualquer pensamento. Essa deselegância frequentemente se revelará lenta ou cara para ser duplicada em outra implementação. Por exemplo, algumas máquinas deixam lixo no registrador do multiplicando após uma multiplicação. A natureza precisa desse lixo acaba por ser parte do *de fato* definição, mas duplicá-la pode impedir o uso de um algoritmo de multiplicação mais rápido.

Finalmente, o uso de uma implementação como uma definição formal é peculiarmente suscetível a confusão quanto ao fato de a descrição em prosa ou a descrição formal ser de fato o padrão. Isso é especialmente verdadeiro para simulações programadas. É preciso também se abster

desde modificações até a implementação enquanto está servindo como um padrão.

Incorporação direta

Uma técnica adorável para disseminar e impor definições está disponível para o arquiteto de sistema de software. É especialmente útil para estabelecer a sintaxe, se não a semântica, das interfaces entre módulos. Essa técnica é projetar a declaração dos parâmetros passados ou armazenamento compartilhado e exigir que as implementações incluam essa declaração por meio de uma operação de tempo de compilação (uma macro ou um% INCLUDE em PL / I). Se, além disso, toda a interface é referenciada apenas por nomes simbólicos, a declaração pode ser alterada adicionando ou inserindo novas variáveis apenas com recompilação, não alteração, do programa em uso.

Conferências e tribunais

Não é preciso dizer que as reuniões são necessárias. As centenas de consultas feitas por homens devem ser complementadas por reuniões maiores e mais formais. Descobrimos que dois níveis desses são úteis. A primeira é uma conferência semanal de meio dia de todos os arquitetos, além de representantes oficiais dos implementadores de hardware e software e planejadores de mercado. O arquiteto-chefe do sistema preside.

Qualquer pessoa pode propor problemas ou mudanças, mas as propostas geralmente são distribuídas por escrito antes da reunião. Um novo problema geralmente é discutido um pouco. A ênfase está na criatividade, e não apenas na decisão. O grupo tenta inventar muitas soluções para os problemas, então algumas soluções são passadas para um ou mais dos arquitetos para detalhamento em propostas de mudança manual formuladas com precisão.

Propostas de mudanças detalhadas são então tomadas para decisões. Eles foram divulgados e cuidadosamente considerados por implementadores e usuários, e os prós e contras estão bem delineados. Se um consenso surgir, muito bem. Caso contrário, o arquiteto-chefe decide. Minutos

são mantidas e as decisões são formalmente, prontamente e amplamente divulgadas.

As decisões das conferências semanais dão resultados rápidos e permitem que o trabalho prossiga. Se alguém é *também* infeliz, apelos instantâneos ao gerente de projeto são possíveis, mas isso acontece muito raramente.

A fecundidade dessas reuniões vem de várias fontes:

1. O mesmo grupo - arquitetos, usuários e implementadores - se reúne semanalmente durante meses. Não é necessário tempo para atualizar as pessoas.
2. O grupo é brilhante, engenho, bem versado nas questões e profundamente envolvido no resultado. Ninguém tem uma função de "consultor". Todos estão autorizados a assumir compromissos vinculativos.
3. Quando os problemas são levantados, as soluções são buscadas dentro e fora dos limites óbvios.
4. A formalidade das propostas escritas concentra a atenção, força a decisão e evita inconsistências elaboradas pelo comitê.
5. A atribuição clara do poder de tomada de decisão ao arquiteto-chefe evita concessões e atrasos.

Com o passar do tempo, algumas decisões não vão bem. Alguns assuntos menores nunca foram aceitos de todo o coração por um ou outro dos participantes. Outras decisões desenvolveram problemas imprevistos e, às vezes, a reunião semanal não concordava em reconsiderá-los. Portanto, há um acúmulo de recursos menores, questões em aberto ou descontentamentos. Para resolver isso, realizamos sessões anuais da suprema corte, durando normalmente duas semanas. (Eu os seguraria a cada seis meses se estivesse fazendo isso de novo.)

Essas sessões foram realizadas imediatamente antes das principais datas de congelamento do manual. Os presentes incluíam não apenas o grupo de arquitetura e os representantes de arquitetura dos programadores e implementadores, mas também os gerentes de programação, marketing e esforços de implementação. O gerente de projeto System / 360 presidiu. A pauta normalmente consistia em cerca de 200 itens, a maioria menores, que eram enumerados em gráficos colocados ao redor da sala. Tudo

lados foram ouvidos e as decisões tomadas. Pelo milagre da edição de texto computadorizada (e muito trabalho excelente da equipe), cada participante encontrou um manual atualizado, incorporando as decisões da manhã de ontem, em seu assento todo.

Esse "festivais de outono" eram úteis não apenas para resolver decisões, mas também para fazer com que fossem aceitas. Todos foram ouvidos, todos participaram, todos entenderam melhor as intrincadas restrições e as inter-relações entre as decisões.

Múltiplas implementações

Os arquitetos do System / 360 tinham duas vantagens quase sem precedentes: tempo suficiente para trabalhar com cuidado e influência política igual à dos implementadores. A provisão de tempo suficiente veio da programação da nova tecnologia; a igualdade política veio da construção simultânea de múltiplas implementações. A necessidade de compatibilidade estrita entre eles serviu como o melhor agente possível de fiscalização das especificações.

Na maioria dos projetos de computador, chega um dia em que se descobre que a máquina e o manual não combinam. Quando ocorre o confronto, o manual geralmente perde, pois pode ser alterado com muito mais rapidez e economia do que a máquina. Não é assim, no entanto, quando há várias implementações. Então, os atrasos e custos associados ao conserto da máquina errante podem ser superados por atrasos e custos na revisão das máquinas que seguiram fielmente o manual.

Essa noção pode ser aplicada com sucesso sempre que uma linguagem de programação está sendo definida. Pode-se ter certeza de que vários interpretadores ou compiladores, mais cedo ou mais tarde, terão que ser construídos para atender a vários objetivos. A definição será mais clara e a disciplina mais rígida se pelo menos duas implementações forem construídas inicialmente.

O registro do telefone

À medida que a implementação prossegue, surgem inúmeras questões de interpretação arquitetônica, não importa o quanto precisa seja a especificação. Obviamente

Muitas dessas questões requerem ampliações e esclarecimentos no texto. Outros apenas refletem mal-entendidos.

É essencial, no entanto, encorajar o implemento intrigado a telefonar para o arquiteto responsável e fazer sua pergunta, em vez de adivinhar e prosseguir. É tão vital reconhecer que as respostas a essas perguntas são *Excathedra* pronunciamentos arquitetônicos que devem ser ditos a todos.

Um mecanismo útil é um *registro de telefone* mantido pelo arquiteto. Nele ele registra cada pergunta e cada resposta. A cada semana, os logs dos vários arquitetos são concatenados, reproduzidos e distribuídos aos usuários e implementadores. Embora esse mecanismo seja bastante informal, é rápido e abrangente.

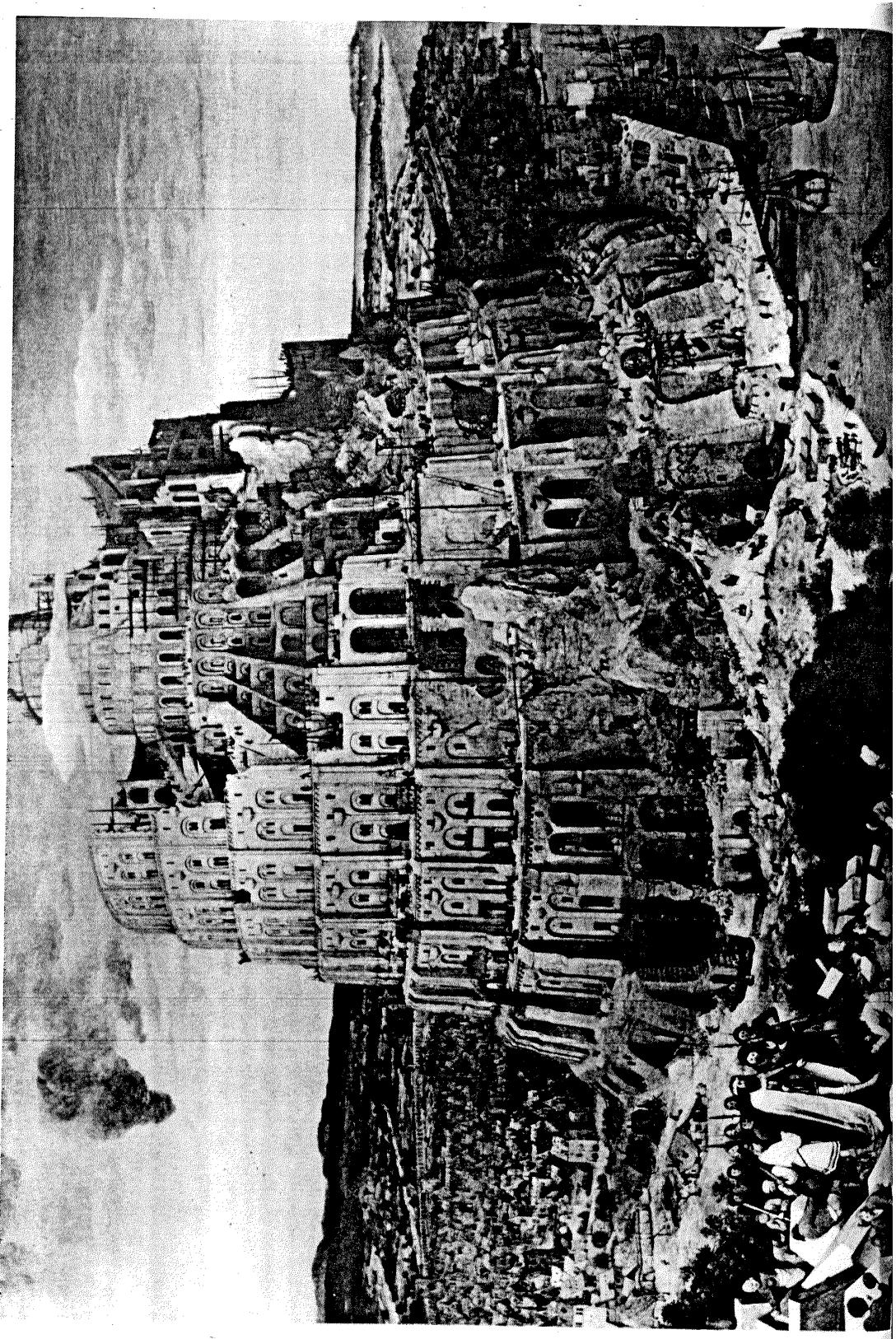
Teste de Produto

O melhor amigo do gerente de projeto é seu adversário diário, a organização independente de teste de produtos. Este grupo verifica máquinas e programas em relação às especificações e atua como um advogado do diabo, identificando todos os defeitos e discrepâncias concebíveis. Cada organização de desenvolvimento precisa de um grupo de auditoria técnica independente para mantê-la honesta.

Em última análise, o cliente é o auditor independente. À luz impiedosa do uso real, cada falha aparecerá. O grupo de teste de produto, então, é o cliente substituto, especializado em encontrar falhas. Vez após vez, o testador de produto cuidadoso encontrará lugares onde a palavra não foi aprovada, onde as decisões de design não foram devidamente compreendidas ou implementadas com precisão. Por esse motivo, esse grupo de teste é um elo necessário na cadeia pela qual a palavra do design é passada, um elo que precisa operar antecipadamente e simultaneamente com o design.

7

*Por que a Torre de
Babel falhou?*



7

Por que a torre de Babel Fail?

Agora, toda a terra usava apenas um idioma, com poucas palavras. Por ocasião de uma migração do leste, os homens descobriram uma planície na terra de Shinar e se estabeleceram lá. Então eles disseram uns aos outros: "Venham, façamos tijolos, que queimemos bem." Então eles usaram tijolos como pedra e betume como argamassa. Então eles disseram: "Vinde, vamos construir para nós mesmos uma cidade com uma torre cujo cume alcance os céus (fazendo assim um nome para nós mesmos), para que não sejamos espalhados por toda a terra." olhe para a cidade e a torre que os seres humanos construíram. O Senhor disse: "Eles são apenas um povo e todos falam a mesma língua. Se isso é o que eles podem fazer no início, então nada do que eles decidam fazer será impossível para eles. Venha, vamos descer, e eles falam tanto que eles não entenderão a fala uns dos outros. Assim o Senhor os dispersou dali por toda a terra, de modo que tiveram que parar de construir a cidade.

GÊNESIS 11: 1-8

P. Breughel, o Velho, "Turmbau zu Babel," 1563
Kunsthistorisches Museum, Viena

74 Por que a Torre de Babel falhou?

Uma Auditoria de Gestão do Projeto Babel

De acordo com o relato de Gênesis, a torre de Babel foi o segundo maior empreendimento de engenharia do homem, depois da arca de Noé. Babel foi o primeiro fiasco da engenharia.

A história é profunda e instrutiva em vários níveis. Vamos, entretanto, examiná-lo puramente como um projeto de engenharia e ver quais lições de gerenciamento podem ser aprendidas. O projeto deles estava bem equipado com os pré-requisitos para o sucesso? Eles tinham:

1. A *missão clara*? Sim, embora ingenuamente impossível. O projeto falhou muito antes de atingir essa limitação fundamental.
 - 2 *Mão de obra*? Muito disso.
 - 3 *Materiais*? Argila e asfalto são abundantes na Mesopotâmia.
 4. *Tempo*? Sim, não há nenhuma sugestão de restrição de tempo.
 5. *Adequado tecnologia*? Sim, a estrutura piramidal ou cônica é inherentemente estável e distribui bem a carga de compressão.
- Claramente, a alvenaria foi bem compreendida. O projeto falhou antes de atingir as limitações tecnológicas.

Bem, se eles tinham todas essas coisas, por que o projeto falhou? Onde eles faltaram? Em dois aspectos - *comunicação*, e seu consequente, *organização*. Eles não conseguiam falar um com o outro; portanto, eles não podiam coordenar. Quando a coordenação falhou, pare de trabalhar. Lendo nas entrelinhas, concluímos que a falta de comunicação levou a disputas, sentimentos ruins e ciúmes de grupo. Logo os clãs começaram a se separar, preferindo o isolamento a disputas.

Comunicação no Grande Projeto de Programação

Assim é hoje. Desastres de programação, desajustes funcionais e bugs de sistema surgem porque a mão esquerda não sabe o que a direita está fazendo. À medida que o trabalho prossegue, as várias equipes mudam lentamente as funções, tamanhos e velocidades de seus próprios programas e, explícita ou implicitamente, mudam suas suposições sobre as entradas disponíveis e os usos a serem feitos das saídas.

Por exemplo, o implementador de uma função de sobreposição de programação pode gerar problemas e reduzir *Rapidez*, basear-se em estatísticas que mostram quão raramente esta função irá surgir na aplicação gramas. Enquanto isso, de volta ao rancho, seu vizinho pode estar projetando uma parte importante do supervisor de modo que dependa criticamente da velocidade dessa função. Essa mudança na velocidade em si torna-se uma grande mudança nas especificações e precisa ser proclamada no exterior e avaliada do ponto de vista do sistema.

Como, então, as equipes devem se comunicar umas com as outras? De todas as maneiras possíveis.

- *Informalmente.* Um bom serviço telefônico e uma definição clara das dependências entre os grupos encorajarão as centenas de ligações das quais depende a interpretação comum de documentos escritos.
- *Encontros.* As reuniões regulares do projeto, com uma equipe após a outra dando instruções técnicas, são inestimáveis. Centenas de pequenos mal-entendidos são eliminados dessa maneira.
- *Livro de exercícios.* Um caderno de trabalho formal do projeto deve ser iniciado no início. Isso merece uma seção própria.

A apostila do projeto

That. A pasta de trabalho do projeto não é tanto um documento separado, mas uma estrutura imposta aos documentos que o projeto produzirá de qualquer maneira.

Tudo os documentos do projeto precisam fazer parte dessa estrutura. Isso inclui objetivos, especificações externas, especificações de interface, padrões técnicos, especificações internas e memorandos administrativos.

Por que. A prosa técnica é quase imortal. Se alguém examina a genealogia de um manual do cliente para uma peça de hardware ou software, pode rastrear não apenas as idéias, mas também muitas das próprias frases e parágrafos até os primeiros memorandos propondo o produto ou explicando o primeiro design. Para o redator técnico, o pastete é tão poderoso quanto a caneta.

Sendo assim, e visto que os manuais de qualidade do produto de amanhã crescerão a partir dos memorandos de hoje, é muito importante obter a estrutura correta da documentação. O design inicial da pasta de trabalho do projeto garante que a estrutura da documentação em si seja elaborada, e não aleatória. Além disso, o estabelecimento de uma estrutura molda posteriormente a escrita em segmentos que se encaixam nessa estrutura.

O segundo motivo para a pasta de trabalho do projeto é o controle da distribuição de informações. O problema não é restringir as informações, mas garantir que as informações relevantes cheguem a todas as pessoas que delas precisam.

O primeiro passo é numerar todos os memorandos, para que listas ordenadas de títulos fiquem disponíveis e cada trabalhador veja se tem o que deseja. A organização da pasta de trabalho vai muito além disso para estabelecer uma estrutura em árvore de memorandos. A estrutura em árvore permite que as listas de distribuição sejam mantidas por subárvore, se isso for desejável.

Mecânica. Como acontece com tantos problemas de gerenciamento de programação, o problema do memorando técnico piora de forma não linear à medida que o tamanho aumenta. Com 10 pessoas, os documentos podem ser simplesmente numerados. Com 100 pessoas, várias sequências lineares geralmente são suficientes. Com 1000, inevitavelmente espalhados por vários locais físicos, o *necessidade* para uma pasta de trabalho estruturada aumenta e o *Tamanho* da pasta de trabalho aumenta. Como então a mecânica deve ser tratada?

Acho que isso foi bem feito no projeto OS / 360. A necessidade de uma pasta de trabalho bem estruturada foi fortemente estimulada por OS Locken, que viu sua eficácia em seu projeto anterior, o sistema operacional 1410-7010.

Nós rapidamente decidimos que *cada* o programador deve ver *lá* o material, ou seja, deve ter uma cópia da apostila em seu próprio escritório.

De importância crítica é a atualização oportuna. A pasta de trabalho deve ser atual. Isso é muito difícil de fazer se documentos inteiros devem ser redigitados para alterações. Em um livro de folhas soltas, no entanto, apenas as páginas precisam ser alteradas. Tínhamos disponível um sistema de edição de texto controlado por computador, e isso provou ser de valor inestimável para manutenção oportuna. Desvio

os masters eram preparados diretamente na impressora do computador e o tempo de resposta era de menos de um dia. O destinatário de todas essas páginas atualizadas tem um problema de assimilação, no entanto. Quando ele recebe pela primeira vez uma página alterada, ele quer saber, "O que foi alterado?" Quando ele o consulta mais tarde, ele quer saber: "Qual é a definição hoje?"

A última necessidade é atendida pelo documento mantido continuamente. O destaque das mudanças requer outras etapas. Primeiro, deve-se marcar o texto alterado na página, por exemplo, por uma barra vertical na margem ao lado de cada linha alterada. Em segundo lugar, é necessário distribuir com as novas páginas um breve resumo de mudanças escrito separadamente que lista as mudanças e comentários sobre seu significado.

Nosso projeto não estava em andamento há seis meses quando encontramos outro problema. A pasta de trabalho tinha cerca de um metro e meio de espessura! Se tivéssemos empilhado as 100 cópias servindo aos programadores em nossos escritórios no edifício Time-Life em Manhattan, elas teriam se destacado acima do próprio edifício. Além disso, a distribuição da mudança diária era em média de duas polegadas, cerca de 150 páginas a serem intercaladas no todo. A manutenção da pasta de trabalho começou a demorar um tempo significativo a cada dia de trabalho.

Nesse ponto, mudamos para microficha, uma mudança que economizou um milhão de dólares, mesmo considerando o custo de uma leitora de microficha para cada escritório. Conseguimos organizar um excelente retorno na produção de microfichas; a pasta de trabalho encolheu de três pés cúbicos para um sexto de um pé cúbico e, mais significativamente, as atualizações apareceram em blocos de cem páginas, reduzindo o problema de interfiling a cem vezes.

A microficha tem suas desvantagens. Do ponto de vista do gerente, a interface estranha das páginas de papel garantiu que as mudanças fossem *leitura*, qual era o objetivo da apostila. A microficha tornaria a manutenção da pasta de trabalho muito fácil, a menos que o arquivo de atualização fosse distribuído com um documento em papel enumerando as alterações.

Além disso, uma microficha não pode ser destacada, marcada e comentada prontamente pelo leitor. Documentos com os quais o leitor tem

interagiram são mais eficazes para o autor e mais úteis para o leitor.

No geral, acho que o microfilme foi um mecanismo muito feliz e eu o recomendaria em vez de uma pasta de trabalho em papel para projetos muito grandes.

Como alguém faria isso hoje? Com a tecnologia de sistema atual disponível, acho que a técnica escolhida é manter a pasta de trabalho no arquivo de acesso direto, marcada com barras de alteração e datas de revisão. Cada usuário o consultaria em um terminal de exibição (as máquinas de escrever são muito lentas). Um resumo da mudança, preparado diariamente, seria armazenado no formato LIFO em um ponto de acesso fixo. O programador provavelmente leria isso diariamente, mas se perdesse um dia, só precisaria ler mais no dia seguinte. Ao ler o resumo da alteração, ele poderia interromper para consultar o próprio texto alterado.

Observe que a pasta de trabalho em si não é alterada. É ainda a montagem de toda a documentação do projeto, estruturada de acordo com um desenho criterioso. A única mudança está na mecânica de distribuição e consulta. DC Engelbart e seus colegas do Stanford Research Institute construíram esse sistema e estão usando-o para construir e manter documentação para a rede ARPA.

DL Parnas da Carnegie-Mellon University propôs um solução ainda mais radical.¹ Sua tese é que o programador é mais eficaz se protegido, em vez de exposto aos detalhes da construção de outras partes do sistema que não as suas. Isso pressupõe que todas as interfaces sejam definidas de forma completa e precisa. Embora esse seja definitivamente um bom design, depende de que sua realização perfeita seja uma receita para o desastre. Um bom sistema de informação expõe erros de interface e estimula sua correção.

Organização no Grande Projeto de Programação

Se houver n trabalhadores em um projeto, existem $(n \cdot n) / 2$ interfaces através das quais pode haver comunicação, e há potencialmente quase 2 "equipes nas quais a coordenação deve ocorrer. O objetivo da organização é reduzir a quantidade de comunicação

e coordenação necessária; portanto, a organização é um ataque radical aos problemas de comunicação tratados acima.

Os meios pelos quais a comunicação é evitada são *divisão de trabalho* e *especialização da função*. A estrutura em forma de árvore das organizações reflete a necessidade cada vez menor de comunicação detalhada quando a divisão e a especialização do trabalho são aplicadas.

Na verdade, uma organização em árvore surge realmente como uma estrutura de autoridade e responsabilidade. O princípio de que nenhum homem pode servir a dois senhores dita que a estrutura de autoridade seja semelhante a uma árvore. Mas a estrutura de comunicação não é tão restrita e a árvore é uma aproximação quase imperceptível da estrutura de comunicação, que é uma rede. As inadequações da aproximação da árvore dão origem a grupos de pessoal, forças-tarefa, comitês e até mesmo a organização do tipo matricial usada em muitos laboratórios de engenharia.

Vamos considerar uma organização de programação semelhante a uma árvore e examinar os fundamentos que qualquer subárvore deve ter para ser eficaz. Eles são:

1. uma missão
2. um produtor
3. um diretor técnico ou arquiteto
4. um cronograma
5. uma divisão de trabalho
6. definições de interface entre as partes

Tudo isso é óbvio e convencional, exceto a distinção entre o produtor e o diretor técnico. Vamos primeiro considerar os dois papéis, depois seu relacionamento.

Qual é o papel do produtor? Ele monta a equipe, divide o trabalho e estabelece o cronograma. Ele adquire e vai adquirindo os recursos necessários. Isso significa que grande parte de sua função é a comunicação fora da equipe, para cima e para os lados. Ele estabelece o padrão de comunicação e relatórios dentro da equipe. Finalmente, ele garante que o cronograma seja cumprido, mudando os recursos e a organização para responder às mudanças nas circunstâncias.

80 Por que a Torre de Babel falhou?

E o diretor técnico? Ele concebe o projeto a ser construído, identifica suas subpartes, especifica como será a aparência do lado de fora e esboça sua estrutura interna. Ele fornece unidade e integridade conceitual para todo o design; assim, ele serve como um limite para a complexidade do sistema. Conforme surgem problemas técnicos individuais, ele inventa soluções para eles ou muda o projeto do sistema conforme necessário. Ele é, na adorável frase de Al Capp, "o homem por dentro que trabalha com gambás". Suas comunicações são principalmente dentro da equipe. Seu trabalho é quase totalmente técnico.

Agora está claro que os talentos necessários para essas duas funções são bastante diferentes. Os talentos vêm em muitas combinações diferentes; e a combinação particular incorporada no produtor e no diretor deve governar o relacionamento entre eles. As organizações devem ser projetadas em torno das pessoas disponíveis; não pessoas encaixadas em organizações de teoria pura.

Três relacionamentos são possíveis e todos os três são encontrados na prática bem-sucedida.

O produtor e o diretor técnico podem ser o mesmo homem.

Isso é facilmente viável em equipes muito pequenas, talvez de três a seis programadores. Em projetos maiores, raramente é viável, por duas razões. Em primeiro lugar, raramente é encontrado o homem com forte talento administrativo e forte talento técnico. Os pensadores são raros; os fazedores são mais raros; e pensadores-realizadores são os mais raros.

Em segundo lugar, em um projeto maior, cada uma das funções é necessariamente um trabalho de tempo integral, ou mais. É difícil para o produtor delegar o suficiente de suas funções para dar-lhe algum tempo técnico. É impossível para o diretor delegar o seu sem comprometer a integridade conceitual do projeto.

O produtor pode ser o chefe, o diretor, seu braço direito.

A dificuldade aqui é estabelecer o diretor *autoridade* para tomar decisões técnicas sem afetar seu tempo, como faria com colocá-lo na cadeia de comando da gerência.

Obviamente, o produtor deve proclamar a autoridade técnica do diretor, e deve apoiá-la em uma proporção extremamente elevada de

os casos de teste que surgirão. Para que isso seja possível, o produtor e o diretor devem ter opiniões semelhantes sobre a filosofia técnica fundamental; eles devem discutir as principais questões técnicas em particular, antes que elas realmente se tornem oportunas; e o produtor deve ter um grande respeito pelas proezas técnicas do diretor.

Menos obviamente, o produtor pode fazer todos os tipos de coisas sutis com os símbolos de status (tamanho do escritório, carpete, móveis, cópias de carbono etc.) para proclamar que o diretor, embora fora da linha de gestão, é uma fonte de poder de decisão.

Isso pode ser feito para funcionar de forma muito eficaz. Infelizmente, raramente é tentado. O trabalho menos bem executado pelos gerentes de projeto é utilizar o gênio técnico que não é forte em talento de gerenciamento.

O diretor pode ser o chefe e o produtor o seu braço direito.

Robert Heinlein, em *O homem que vendeu a lua*, descreve tal arranjo em um gráfico, por exemplo:

Coster enterrou o rosto nas mãos e ergueu os olhos. "Eu sei disso. Eu sei o que precisa ser feito - mas toda vez que tento resolver um problema técnico, algum idiota quer que eu tome uma decisão sobre caminhões - ou telefones - ou alguma coisa maldita. Sinto muito, Sr. Harriman. Eu pensei que poderia fazer isso."

Harriman disse muito gentilmente: "Não se deixe confundir. Bob. Você não dormiu muito ultimamente, não é? É o seguinte - we'll superou um rápido em Ferguson. Vou pegar aquela mesa em que você está por alguns dias e construir uma estrutura para protegê-la contra essas coisas. Quero aquela sua cabeça pensando sobre vetores de reação e eficiência de combustível e tensões de design, não sobre contratos para caminhões. Harriman foi até a porta, olhou ao redor da sala externa e avistou um homem que poderia ou não ser o secretário-chefe do escritório. - Ei, você! Vem cá."

O homem pareceu assustado, levantou-se, veio até a porta e disse: "Sim?"

"Eu quero aquela mesa no canto e todas as coisas que estão nela movidas para um escritório vazio neste andar, imediatamente."

82 Por que a Torre de Babel FaU?

Ele supervisionou a mudança de Coster e sua outra mesa para outro escritório, providenciou para que o telefone do novo escritório fosse desconectado e, pensando bem, mandou mudar um sofá para lá também. "Vamos instalar um projetor, uma máquina de desenho, estantes de livros e outras coisas assim esta noite", disse ele a Coster. "Basta fazer uma lista de tudo o que você precisa - trabalhar em Engenharia. " Ele voltou ao escritório do engenheiro-chefe nominal e começou a trabalhar alegremente tentando descobrir onde estava a organização e o que havia de errado com ela.

Cerca de quatro horas depois, ele levou Berkeley para encontrar Coster. O engenheiro-chefe estava dormindo em sua mesa, a cabeça apoiada nos braços. Harriman começou a recuar, mas Coster despertou. "Oh! Desculpe," ele disse, corando, "Eu devo ter cochilado."

"É por isso que trouxe o sofá para você", disse Harriman. "É mais tranquilo. Bob, conheça Jock Berkeley. Ele é seu novo escravo. Você continua sendo o engenheiro-chefe e chefe indiscutível. Jock é LordHigh, tudo o mais. De agora em diante, você não tem absolutamente nada com que se preocupar - exceto pelo pequeno detalhe de construir uma nave lunar. "

Eles apertaram as mãos. "Só uma coisa eu peço, Sr. Coster", Berkeley disse sério, "ignore-me o quanto quiser - você terá que dirigir o show técnico - mas, pelo amor de Deus, registre para que eu saiba o que está acontecendo. Vou colocar um interruptor na sua mesa para operar um gravador lacrado na minha mesa. "

"Multar!" Coster estava parecendo, Harriman pensou, já mais jovem.

"E se você quiser algo que não seja técnico, não faça você mesmo. Basta girar um botão e assobiar; isso será feito!" Berkeley olhou para Harriman. "O chefe disse que quer falar com você sobre o verdadeiro trabalho. Vou deixá-lo e me ocupar." Ele saiu.

Harriman sentou-se; Coster fez o mesmo e disse: "Uau!" "Sentir-se melhor?"

"Gosto da aparência daquele tal de Berkeley."

*"Isso é bom; ele é seu irmão gêmeo de agora em diante. Pare de se preocupar; eu já usei ele antes. Você pensa que está morando em um hospital bem administrado." **

Este relato dificilmente precisa de qualquer comentário analítico. Esse arranjo também pode funcionar de maneira eficaz.

Suspeito que o último arranjo seja melhor para equipes pequenas, conforme discutido no Capítulo 3, "A equipe cirúrgica". Acho que o produtor como chefe é um arranjo mais adequado para as subárvore maiores de um projeto realmente grande.

A Torre de Babel foi talvez o primeiro fiasco da engenharia, mas não foi o último. A comunicação e sua consequente organização são fundamentais para o sucesso. As técnicas de comunicação e organização exigem do gerente muito pensamento e tanta competência experiente quanto a própria tecnologia de software.

8

Chamando o tiro



8

Chamando o tiro

A prática é o melhor de todos os instrutores.

PUBUUUS

A experiência é uma boa professora, mas os tolos não aprenderão de outra maneira.

POOR RICHARD'S ALMANAC

Douglass Crockwell, "Ruth calls his shot", World Series, 1932
Reproduzido com permissão da Esquire Magazine e Douglass Crockwell, © 1945
(renovado em 1973) pela Esquire, Inc. e cortesia do National Baseball Museum.

Quanto tempo levará um trabalho de programação do sistema? Quanto esforço será necessário? Como fazer uma estimativa?

Já sugeri proporções que parecem se aplicar ao planejamento de tempo, codificação, teste de componente e teste de sistema. Primeiro, deve-se dizer que *não* estimar a tarefa inteira estimando apenas a porção de codificação e, em seguida, aplicando as razões. A codificação é apenas um sexto ou mais do problema, e erros em sua estimativa ou nas proporções podem levar a resultados ridículos.

Em segundo lugar, deve-se dizer que os dados para a construção de pequenos programas isolados não são aplicáveis a produtos de sistemas de programação. Para um programa com média de cerca de 3.200 palavras, por exemplo, Sackman, Erikson e Grant relatam um tempo médio de código mais depuração de cerca de 178 horas para um único programador, um número que extrapolaria para dar uma produtividade anual de 35.800 declarações por ano. Um programa com metade desse tamanho demorava menos de um quarto do tempo e a produtividade extrapolada é de quase 80.000 declarações por ano.¹ Devem ser adicionados os tempos de planejamento, documentação, teste, integração de sistema e treinamento. A extração linear de tais figuras de sprint não tem sentido. Extração de tempos para o traço de cem metros mostra que um homem pode correr uma milha em menos de três minutos.

Antes de descartá-los, no entanto, observemos que esses números, embora não sejam para problemas estritamente comparáveis, sugerem que o esforço vale como uma potência de tamanho. *até* quando nenhuma comunicação está envolvida, exceto a de um homem com suas memórias.

A Figura 8.1 conta a triste história. Ele ilustra os resultados relatados de um estudo feito por Nanus e Farr² na System Development Corporation. Isso mostra um expoente de 1,5; isso é,

$$\text{esforço} = (\text{constante}) \times (\text{número de instruções})^{1,5}$$

Outro estudo SDC relatado por Weinwurm³ também mostra um expoente próximo a 1,5.

Alguns estudos sobre a produtividade do programador foram feitos e várias técnicas de estimativa foram propostas. Morin preparou um levantamento dos dados publicados.⁴ Aqui, darei apenas alguns itens que parecem especialmente esclarecedores.

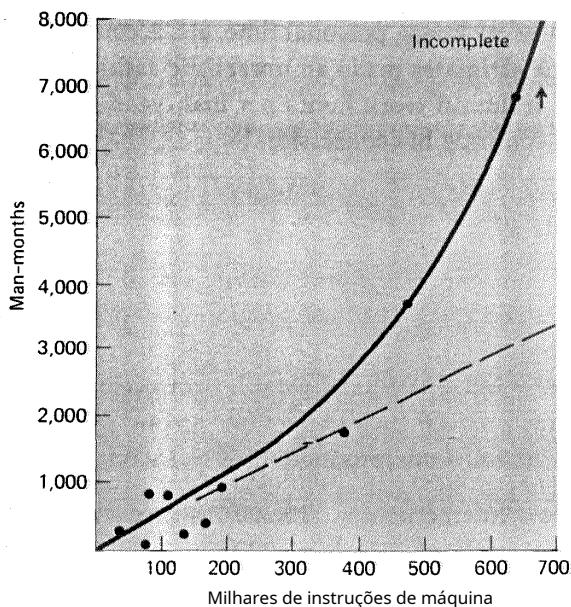


Fig. 8.1 Esforço de programação em função do tamanho do programa

Dados de Portman

Charles Portman, gerente da Divisão de Software da ICL, Computer Equipment Organization (Northwest) em Manchester, oferece outra visão pessoal útil.⁵

Ele descobriu que suas equipes de programação perdiam cronogramas pela metade - cada trabalho estava demorando aproximadamente o dobro do tempo estimado. As estimativas foram muito cuidadosas, feitas por equipes experientes que estimam horas-homem para várias centenas de subtarefas em um gráfico PERT. Quando o padrão de deslizamento apareceu, ele pediu que eles mantivessem registros diários cuidadosos do uso do tempo. Isso mostrou que o erro de estimativa poderia ser inteiramente explicado pelo fato de que suas equipes estavam percebendo apenas 50% da semana de trabalho como programação real e tempo de depuração. Tempo de inatividade da máquina, trabalhos curtos não relacionados de alta prioridade, reuniões, papelada,

negócios, doença, tempo pessoal, etc. responsável pelo resto. Em suma, as estimativas faziam uma suposição irreal sobre o número de horas de trabalho técnico por homem-ano. Minha própria experiência confirma totalmente sua conclusão.⁶

Dados de Aron

Joel Aron, gerente de Tecnologia de Sistemas da IBM em Gaithersburg, Maryland, estudou a produtividade do programador ao trabalhar em nove grandes sistemas (resumidamente, *ampla* significa mais de 25 programadores e 30.000 instruções de entrega).⁷ Ele divide esses sistemas de acordo com as interações entre os programadores (e partes do sistema) e encontra as produtividades da seguinte forma:

Muito poucas interações	10.000 instruções por homem-ano
Algumas interações	5.000
Muitas interações	1.500

Os anos-homem não incluem atividades de suporte e teste de sistema, apenas design e programação. Quando esses números são diluídos por um fator de dois para cobrir o teste do sistema, eles se aproximam dos dados de Harr.

Dados de Harr

John Hair, gerente de programação do Sistema de Comutação Eletrônica da Bell Telephone Laboratories, relatou sua experiência e a de outros em um artigo na Spring Joint Computer Conference de 1969.⁸ Esses dados são mostrados nas Figs. 8,2, 8,3 e 8,4.

Destas, a Fig. 8.2 é a mais detalhada e a mais útil. Os primeiros dois trabalhos são basicamente programas de controle; os dois segundos são basicamente tradutores de idiomas. A produtividade é declarada em termos de palavras depuradas por homem-ano. Isso inclui programação, teste de componentes e teste de sistema. Não está claro quanto do esforço de planejamento, ou esforço de suporte da máquina, escrita e similares, está incluído.

	Prog. unidades	Número de programadores	Anos	Maa- anos	Programa palavras	Palavras /' man-yf
Operacional	cinquenta	53	4	101	52, OQG	515
Manutenção	36	60	4	81	51.000	630
Compilador	13	9	• 2'A	1?	38.000	
Tradutor (Montador de dados)	quinze	13	2%	onze	25.000	2270

Fig. 8.2 Resumo de quatro No. 1 Empregos do programa ESS

As produtividades também se enquadram em duas classificações; aqueles para programas de controle são cerca de 600 palavras por homem-ano; aqueles para tradutores são cerca de 2.200 palavras por homem-ano. Observe que todos os quatro programas são de tamanho semelhante - a variação é no tamanho dos grupos de trabalho, duração e número de módulos. O que é causa e o que é efeito? Os programas de controle exigiam mais pessoas porque eram mais complicados? Ou eles exigiram mais módulos e mais homens-mês porque receberam mais pessoas? Eles demoraram mais devido à maior complexidade ou porque mais pessoas foram designadas? Não se pode ter certeza. Os programas de controle eram certamente mais complexos. Deixando essas incertezas de lado, os números descrevem as produtividades reais alcançadas em um grande sistema, usando as técnicas de programação atuais.

As Figuras 8.3 e 8.4 mostram alguns dados interessantes sobre as taxas de programação e depuração em comparação com as taxas previstas.

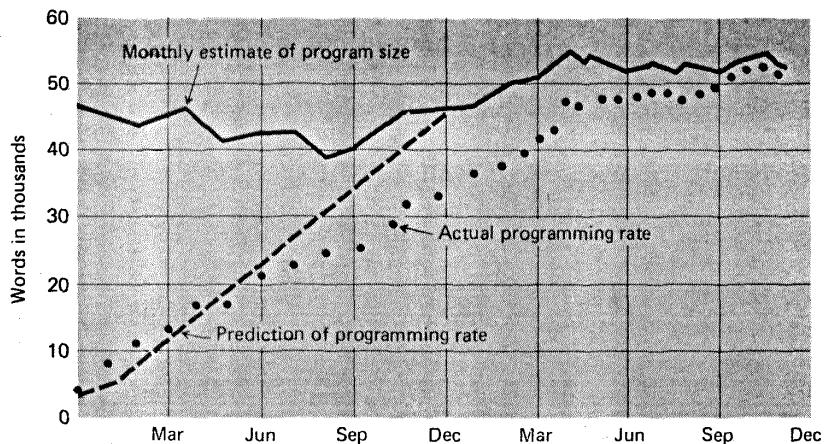


Fig. 8.3 Taxas de programação previstas e reais do ESS

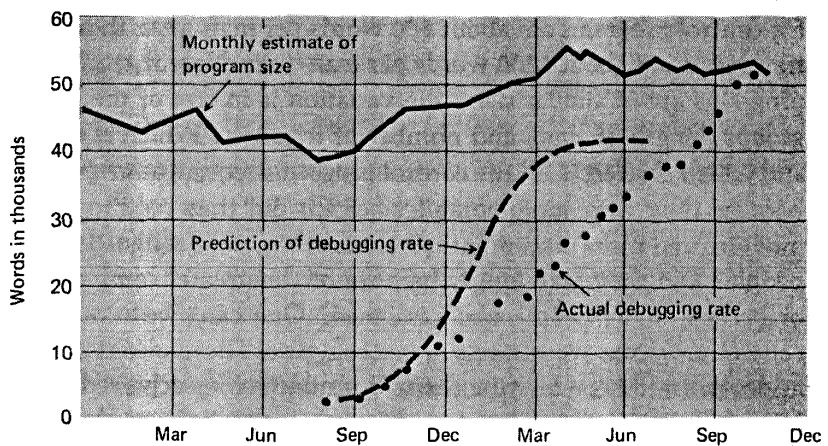


Fig. 8.4 Taxas de depuração previstas e reais do ESS

Dados OS / 360

A experiência do IBM OS / 360, embora não esteja disponível nos detalhes dos dados da Hair, confirma isso. Produtividades na faixa de 600-800 instruções depuradas por homem-ano foram experimentadas por grupos de programas de controle. As produtividades nas instruções depuradas de 2000-3000 por homem-ano foram alcançadas por grupos de tradutores de idiomas. Isso inclui o planejamento feito pelo grupo, teste de componente de codificação, teste de sistema e algumas atividades de suporte. Eles são comparáveis aos dados de Han, pelo que posso dizer.

Os dados de Aron, os dados do Hair e os dados do OS / 360 confirmam diferenças marcantes na produtividade relacionadas à complexidade e dificuldade da própria tarefa. Minha orientação no pântano de estimar a complexidade é que os compiladores são três vezes mais ruins que os programas de aplicativos em lote normais, e os sistemas operacionais são três vezes mais ruins que os compiladores.

9

Dados de Corbato

Os dados do Hair e os dados do OS / 360 são para programação em linguagem assembly. Poucos dados parecem ter sido publicados sobre a produtividade da programação do sistema usando linguagens de nível superior. Corbato, do Projeto MAC do MIT, relata, no entanto, uma produtividade média de 1200 linhas de declarações PL / I depuradas por homem-ano no sistema MULTICS (entre 1 e 2 milhões de palavras).¹⁰

Este número é muito emocionante. Como os demais projetos, o MULTICS inclui programas de controle e tradutores de idiomas. Como as demais, está produzindo um produto de programação de sistema, testado e documentado. Os dados parecem ser comparáveis em termos de tipo de esforço incluído. E o número da produtividade é uma boa média entre as produtividades do programa de controle e do tradutor de outros projetos.

Mas o número de Corbato é *linhas* por homem-ano, não *palavras*/Cada declaração em seu sistema corresponde a cerca de três a cinco palavras de código escrito à mão! Isso sugere duas conclusões importantes.

94 Chamando o tiro

- A produtividade parece constante em termos de declarações elementares, uma conclusão que é razoável em termos do pensamento que uma declaração requer e os erros que pode incluir. "
- A produtividade da programação pode ser aumentada em até cinco vezes quando uma linguagem de alto nível adequada é usada.¹⁸

9

Tem libras

em um saco de cinco libras



9

Dez libras em um saco de cinco libras

O autor deve olhar para Noah e . . . aprender, como fizeram na Arca, a aglomerar uma grande quantidade de matéria em uma bússola muito pequena.

SYDNEY SMITH. REVISÃO DE EDINBURGH

**Gravado_de uma pintura de Heywood Hardy
The Bettman Archive**

Espaço do programa como custo

Quão grande é isso? Além do tempo de execução, o espaço ocupado por um programa é um custo principal. Isso é verdade mesmo para programas proprietários, onde o usuário paga ao autor uma taxa que é essencialmente uma parcela do custo de desenvolvimento. Considere o sistema de software interativo IBM APL. Ele é alugado por US \$ 400 por mês e, quando usado, leva pelo menos 160 K bytes de memória. Em um modelo 165, a memória é alugada por cerca de US \$ 12 por kilobyte por mês. Se o programa estiver disponível em tempo integral, paga-se \$ 400 de aluguel de software e \$ 1920 de aluguel de memória para usar o programa. Se alguém usar o sistema APL apenas quatro horas por dia, os custos são \$ 400 de aluguel de software e \$ 320 de aluguel de memória por mês.

Freqüentemente ouve-se o horror de que uma máquina de 2 M bytes pode ter 400 K dedicados ao seu sistema operacional. Isso é tão tolo quanto criticar um Boeing 747 porque ele custa US \$ 27 milhões. É preciso também perguntar: "O que isso faz?" O que se ganha em facilidade de uso e desempenho (por meio da utilização eficiente do sistema) pelo dinheiro gasto? Será que os \$ 4.800 por mês assim investidos no aluguel de memória teriam sido gastos de forma mais proveitosa em outro hardware, para programadores, para programas de aplicativos?

O designer do sistema coloca parte de seu recurso total de hardware na memória do programa residente quando pensa que fará mais pelo usuário nessa forma do que como somadores, discos, etc. Fazer o contrário seria grosseiramente irresponsável. E o resultado deve ser julgado como um todo. Ninguém pode criticar o tamanho de um sistema de programação *per se* e, ao mesmo tempo, defendem consistentemente uma integração mais próxima do design de hardware e software.

Uma vez que o tamanho é uma grande parte do custo do usuário de um produto de sistema de programação, o construtor deve definir metas de tamanho, controlar o tamanho e desenvolver técnicas de redução de tamanho, assim como o construtor de hardware define metas de contagem de componentes, controla a contagem de componentes e concebe técnicas de redução de contagem. Como qualquer custo, o tamanho em si não é ruim, mas o tamanho desnecessário é.

Controle de tamanho

Para o gerente de projeto, o controle de tamanho é em parte um trabalho técnico e em parte gerencial. É preciso estudar os usuários e seus aplicativos para definir os tamanhos dos sistemas a serem oferecidos. Em seguida, esses sistemas devem ser subdivididos, e cada componente deve ter uma meta de tamanho. Uma vez que as compensações entre tamanho e velocidade ocorrem em grandes saltos quânticos, definir metas de tamanho é um negócio complicado, exigindo conhecimento das compensações disponíveis dentro de cada peça. O gerente sábio também guarda um gatinho para ser alocado à medida que o trabalho prossegue.

No OS / 360, embora tudo isso tenha sido feito com muito cuidado, outras lições ainda tiveram que ser dolorosamente aprendidas.

Primeiro, definir metas de tamanho para o núcleo não é suficiente; é preciso orçar todos os aspectos do tamanho. Na maioria dos sistemas operacionais anteriores, a residência do sistema ficava na fita, e os longos tempos de busca da fita significavam que ninguém ficava tentado a usá-la casualmente para inserir segmentos do programa. O OS / 360 era residente em disco, como seus predecessores imediatos, o Stretch Operating System e o 1410-7010 Disk Operating System. Seus construtores se regozijaram com a liberdade de acessos de disco baratos. O resultado inicial foi desastroso para o desempenho.

Ao definir os tamanhos principais de cada componente, não definimos simultaneamente os orçamentos de acesso. Como qualquer um com uma visão retrospectiva de 20-20 poderia esperar, um programador que encontrou seu programa ultrapassando seu objetivo principal o quebrou em camadas. Esse processo por si só aumentou o tamanho total e tornou a execução mais lenta. Mais seriamente, nosso sistema de controle de gestão não mediou nem detectou isso. Cada homem relatou quanto *essencial*/ele estava usando e, como estava dentro do alvo, ninguém se preocupou.

Felizmente, chegou um dia no esforço em que o simulador de desempenho do OS / 360 começou a funcionar. O primeiro resultado indicou problemas profundos. Fortran H, em um modelo 65 com bateria, simulou a compilação a cinco afirmações por minuto! Aproximando-se mostrou que cada um dos módulos do programa de controle estava fazendo

muitos, muitos acessos ao disco. Até mesmo os módulos de supervisor de alta frequência faziam muitas viagens para o poço, e o resultado era bastante análogo ao debulhamento de páginas.

A primeira moral é clara: Definir *tota*/orçamentos de tamanho, bem como orçamentos de espaço residente; definir orçamentos em acessos de armazenamento de apoio, bem como em tamanhos.

A próxima lição foi muito semelhante. Os orçamentos de espaço foram definidos antes que alocações funcionais precisas fossem feitas para cada módulo. Como resultado, qualquer programador com problemas de tamanho examinava seu código para ver o que ele poderia jogar por cima da cerca no espaço de um vizinho. Assim, os buffers gerenciados pelo programa de controle passaram a fazer parte do espaço do usuário. Mais seriamente, o mesmo acontecia com todos os tipos de bloqueios de controle, e o efeito era totalmente comprometedor da segurança e proteção do sistema.

Portanto, a segunda moral também é clara: defina exatamente o que um módulo deve fazer quando você especifica o quanto grande ele deve ser.

Uma terceira e mais profunda lição mostra por meio dessas experiências. O projeto era grande o suficiente e a comunicação de gerenciamento pobre o suficiente para fazer com que muitos membros da equipe se vissem como competidores ganhando pontos, em vez de construtores fazendo produtos de programação. Cada um subotimizou sua peça para atingir seus objetivos; poucos pararam para pensar sobre o efeito total sobre o cliente. Essa falha na orientação e na comunicação é um grande risco para grandes projetos. Durante toda a implementação, os arquitetos de sistema devem manter vigilância contínua para garantir a integridade contínua do sistema. Além desse mecanismo de policiamento, entretanto, está a questão da atitude dos próprios implementadores. Promover uma atitude de sistema total e orientada para o usuário pode muito bem ser a função mais importante do gerente de programação.

Técnicas Espaciais

Nenhuma quantidade de orçamento e controle de espaço pode tornar um programa pequeno. Isso requer invenção e habilidade.

Obviamente, mais função significa mais espaço, mantendo a velocidade constante. Portanto, a primeira área de habilidade é trocar a função por tamanho. Aqui vem uma questão política inicial e profunda. Quanto dessa escolha deve ser reservado para o usuário? Pode-se projetar um programa com muitos recursos opcionais, cada um ocupando um pouco de espaço. Pode-se projetar um gerador que escolherá a opção Jist e adaptará um programa a ela. Mas, para qualquer conjunto específico de opções, um programa mais monolítico ocuparia menos espaço. É como um carro; se a luz do mapa, o acendedor de cigarros e o relógio tiverem os preços juntos como uma única opção, o pacote custará menos do que se pudéssemos escolher cada um separadamente. Portanto, o designer deve decidir quão refinada será a escolha de opções do usuário.

Ao projetar um sistema para uma variedade de tamanhos de memória, surge outra questão básica. Um efeito limitador impede que a faixa de adequação seja arbitrariamente ampla, mesmo com modularidade de função refinada. No menor sistema, a maioria dos módulos será sobreposta. Uma parte substancial do espaço residente do menor sistema deve ser reservada como uma área temporária ou de paging na qual outras partes são buscadas. O tamanho disso determina o tamanho de todos os módulos. E dividir as funções em pequenos módulos custa desempenho e espaço. Portanto, um grande sistema, que pode permitir uma área transitória vinte vezes maior, só economiza acessos por isso. Ainda sofre com a velocidade e o espaço porque o tamanho do módulo é muito pequeno. Este efeito limita o sistema eficiente máximo que pode ser gerado a partir dos módulos de um sistema pequeno.

A segunda área de habilidade são as compensações espaço-tempo. Para uma determinada função, quanto mais espaço, mais rápido. Isso é verdade em uma faixa surpreendentemente grande. É esse fato que viabiliza a definição de orçamentos espaciais.

O gerente pode fazer duas coisas para ajudar sua equipe a fazer bons compromissos de espaço-tempo. Uma é garantir que eles sejam treinados em técnica de programação, e não apenas confiados na inteligência nativa e na experiência anterior. Para uma nova linguagem ou máquina, isso é especialmente importante. As peculiaridades de seu uso habilidoso precisam ser

aprendeu rapidamente e compartilhou amplamente, talvez com prêmios especiais ou elogios para novas técnicas.

A segunda é reconhecer que a programação tem uma tecnologia e os componentes precisam ser fabricado. Todo projeto precisa de um caderno cheio de boas sub-rotinas ou macros para enfileiramento, pesquisa, hash e classificação. Para cada uma dessas funções, o notebook deve ter pelo menos dois programas, o quick e o squeezed. O desenvolvimento de tal tecnologia é uma tarefa de realização importante que pode ser feita em paralelo com a arquitetura do sistema.

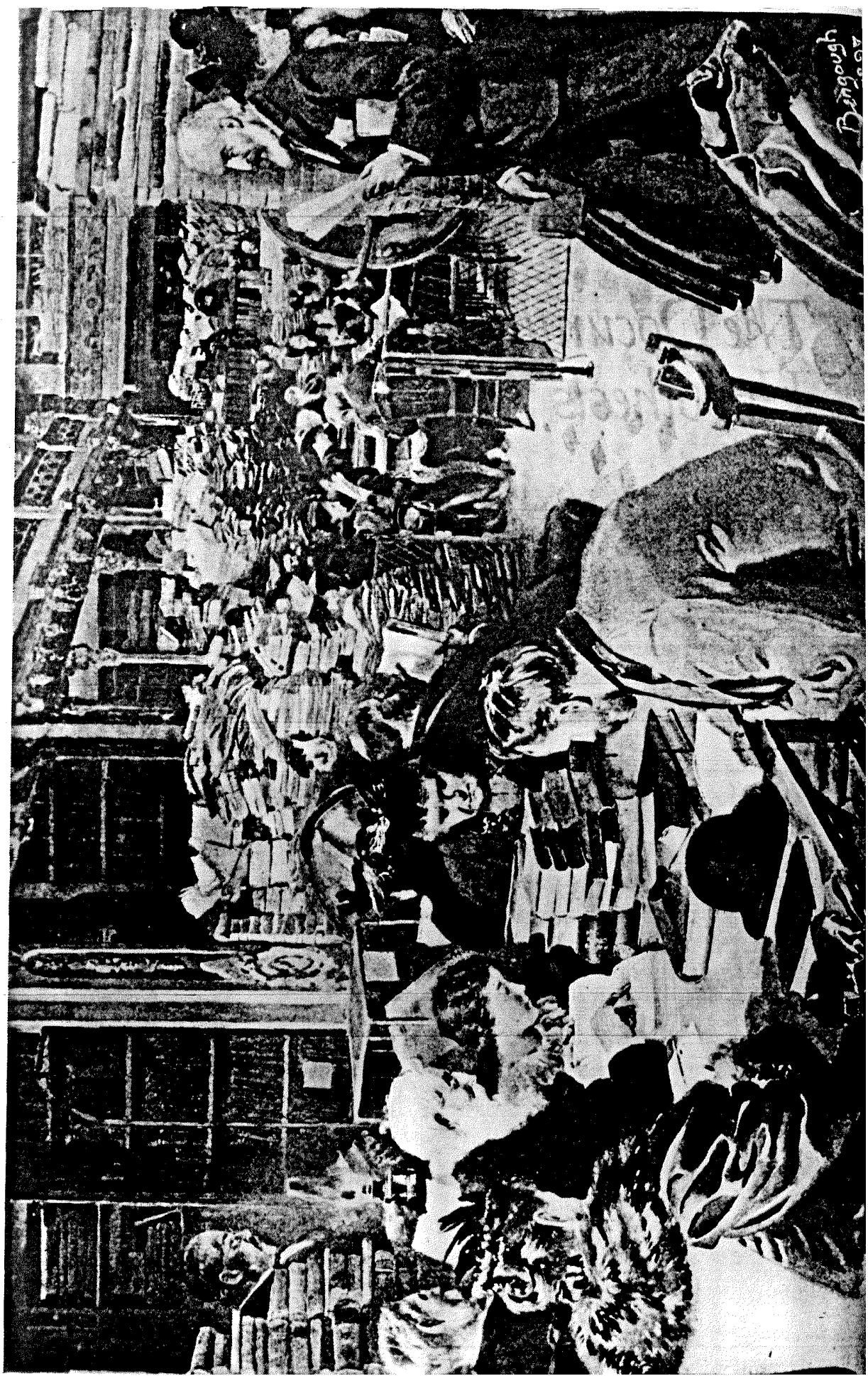
Representação é a essência da programação

Além do artesanato, está a invenção, e é aqui que os programas enxutos, simples e rápidos nascem. Quase sempre, isso é o resultado de um avanço estratégico, e não de uma inteligência tática. Às vezes, o avanço estratégico será um novo algoritmo, como a Transformada Fast Fourier de Cooley-Tukey ou a substituição de um n registro n classificar para um $n * \text{conjunto de comparações}$.

Com muito mais frequência, o avanço estratégico virá ao refazer a representação dos dados ou tabelas. É aqui que reside o cerne de um programa. Mostre-me seus fluxogramas e esconda suas tabelas, e continuarei a ficar perplexo. Mostre-me suas tabelas e geralmente não vou precisar de seus fluxogramas; eles serão óbvios.

É fácil multiplicar exemplos do poder das representações. Lembro-me de um jovem tentando construir um elaborado intérprete de console para um IBM 650. Ele acabou colocando-o em uma quantidade incrivelmente pequena de espaço ao construir um intérprete para o intérprete, reconhecendo que as interações humanas são lentas e raras, mas o espaço era caro. O pequeno e elegante compilador Fortran de Digitek usa uma representação especializada e muito densa para o código do compilador em si, de forma que o armazenamento externo não é necessário. Esse tempo perdido na decodificação dessa representação é recuperado dez vezes mais, evitando-se a entrada-saída. (Os exercícios no final do Capítulo 6 em Brooks e Iverson, *Processamento Automático de Dados*, incluiu uma coleção de exemplos, assim como muitos dos exercícios de Knuth.²⁾

O programador maluco por falta de espaço pode muitas vezes fazer melhor se desvencilhando de seu código, recuando e contemplando seus dados. Representação é a essência da programação.



Dough
02

10

O documentário Hipótese

A hipótese:

Em meio a uma lavagem de papel, um pequeno número de documentos torna-se os eixos críticos em torno dos quais gira todo o gerenciamento de projeto. Essas são as principais ferramentas pessoais do gerente.

W. Bengough, "Scene in the old Congressional Library", 1897 The Bettman Archive

A tecnologia, a organização circundante e as tradições do ofício conspiram para definir certos itens da papelada que um projeto deve preparar. Para o novo gerente, que acabou de trabalhar como artesão, isso parece um incômodo absoluto, uma distração desnecessária e uma maré branca que ameaça engoli-lo. E, de fato, a maioria deles é exatamente isso.

Aos poucos, porém, ele vai percebendo que certo pequeno conjunto desses documentos incorpora e expressa muito de seu trabalho de gestão. A preparação de cada um serve como uma grande ocasião para focar o pensamento e cristalizar discussões que, de outra forma, vagariam sem fim. Sua manutenção torna-se seu mecanismo de vigilância e alerta. O próprio documento serve como uma lista de verificação, um controle de status e um banco de dados para seus relatórios.

Para ver como isso deve funcionar para um projeto de software, vamos examinar os documentos específicos úteis em outros contextos e ver se surge uma generalização.

Documentos para um produto de computador

Suponha que alguém esteja construindo uma máquina. Quais são os documentos críticos?

Objetivos. Isso define a necessidade a ser cumprida e os objetivos, desideratos, restrições e prioridades.

Especificações. Este é um manual de computador mais especificações de desempenho. É um dos primeiros documentos gerados na proposta de um novo produto, e o último documento finalizado.

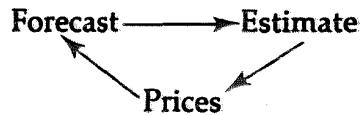
Cronograma

Despesas. Não apenas uma restrição, o orçamento é um dos documentos mais úteis do gerente. A existência de orçamento força decisões técnicas que de outra forma seriam evitadas; e, mais importante, força e esclarece as decisões políticas.

Organograma

Alocações de espaço

Estimativa, previsão, preços. Esses três têm intertravamento cílico, que determina o sucesso ou o fracasso do projeto:



Para uma previsão de mercado, necessita-se de especificações de desempenho e preços postulados. As quantidades da previsão combinam-se com as contagens de componentes do projeto para determinar a estimativa de custo de fabricação e determinam a parcela de desenvolvimento por unidade e os custos fixos. Esses custos, por sua vez, determinam os preços.

Se os preços forem *abaixo* esses postulados, começa uma espiral alegre de sucesso. As previsões aumentam, os custos unitários caem e os preços caem ainda mais.

Se os preços forem *acima* de aqueles postulados, uma espiral desastrosa começa, e todas as mãos devem lutar para quebrá-la. O desempenho deve ser reduzido e novos aplicativos desenvolvidos para suportar previsões maiores. Os custos devem ser reduzidos para produzir estimativas mais baixas. O estresse desse ciclo é uma disciplina que muitas vezes evoca o melhor trabalho do profissional de marketing e do engenheiro.

Também pode causar vacilação ridícula. Lembro-me de uma máquina cujo contador de instruções entrava ou saía da memória a cada seis meses durante um ciclo de desenvolvimento de três anos. Em uma fase, um pouco mais de desempenho seria necessário, então o contador de instruções foi implementado em transistores. Na fase seguinte, a redução de custos era o tema, para que o contador fosse implementado como um local de memória. Em outro projeto, o melhor gerente de engenharia que já vi servia frequentemente como um volante gigante, sua inércia amortecendo as flutuações que vinham do mercado e do pessoal administrativo.

Documentos para um Departamento Universitário

Apesar das imensas diferenças de propósito e atividade, um número semelhante de documentos semelhantes formam o conjunto crítico para o

presidente de um departamento universitário. Quase toda decisão do reitor, reunião do corpo docente ou presidente é uma especificação ou alteração destes documentos:

Objetivos

Descrições do curso

Requisitos de graduação

Propostas de pesquisa (portanto, planos, quando financiados) Programa de aulas e atribuições de ensino

Orçamento

Alocação de espaço

Atribuição de funcionários e alunos de pós-graduação

Observe que os componentes são muito semelhantes aos do projeto de computador: objetivos, especificações do produto, alocações de tempo, alocação de dinheiro, alocação de espaço e alocação de pessoal. Apenas os documentos de preços estão faltando; aqui a legislatura faz essa tarefa. As semelhanças não são acidentais - as preocupações de qualquer tarefa de gerenciamento são o que, quando, quanto, onde e quem.

Documentos para um projeto de software

Em muitos projetos de software, as pessoas começam realizando reuniões para debater a estrutura; então eles começam a escrever programas. Por menor que seja o projeto, entretanto, o gerente deve começar a formalizar imediatamente pelo menos minidocumentos para servir de banco de dados. E ele produz documentos em tons semelhantes aos de outros gerentes.

O quê: objetivos. Isso define a necessidade a ser cumprida e os objetivos, desideratos, restrições e prioridades.

O quê: especificações do produto. Isso começa como uma proposta e acaba como manual e documentação interna. As especificações de velocidade e espaço são uma parte crítica.

Quando: horário

Quanto: orçamento

Onde: alocação de espaço

Quem: organograma. Isso se confunde com a especificação da interface, como prevê a Lei de Conway: "Organizações que projetam sistemas são constrangidas a produzir sistemas que são cópias das estruturas de comunicação dessas organizações."¹ Conway prossegue apontando que o organograma refletirá inicialmente o primeiro design do sistema, que quase certamente não é o correto. Para que o design do sistema seja livre para mudanças, a organização deve estar preparada para mudanças.

Por que ter documentos formais?

Em primeiro lugar, é essencial anotar as decisões. Somente quando se escreve as lacunas aparecem e as inconsistências se projetam. O ato de escrever acaba exigindo centenas de mini-decisões, e é a existência delas que distingue políticas claras e exatas de nebulosas. **ones.**

Em segundo lugar, os documentos comunicarão as decisões a outras pessoas. O gerente ficará continuamente surpreso com o fato de que as políticas que ele adotou para o conhecimento comum são totalmente desconhecidas por algum membro de sua equipe. Como sua função fundamental é manter todos na mesma direção, sua principal tarefa diária será a comunicação, não a tomada de decisões, e seus documentos aliviarão imensamente essa carga.

Finalmente, os documentos de um gerente fornecem a ele um banco de dados e uma lista de verificação. Ao revisá-los periodicamente, ele vê onde está e que mudanças de ênfase ou mudanças de direção são necessárias.

Não compartilho a visão projetada pelo vendedor do "sistema de gerenciamento de informação total", em que o executivo faz uma pesquisa em um computador e uma tela mostra sua resposta. Existem muitas razões fundamentais pelas quais isso nunca acontecerá.

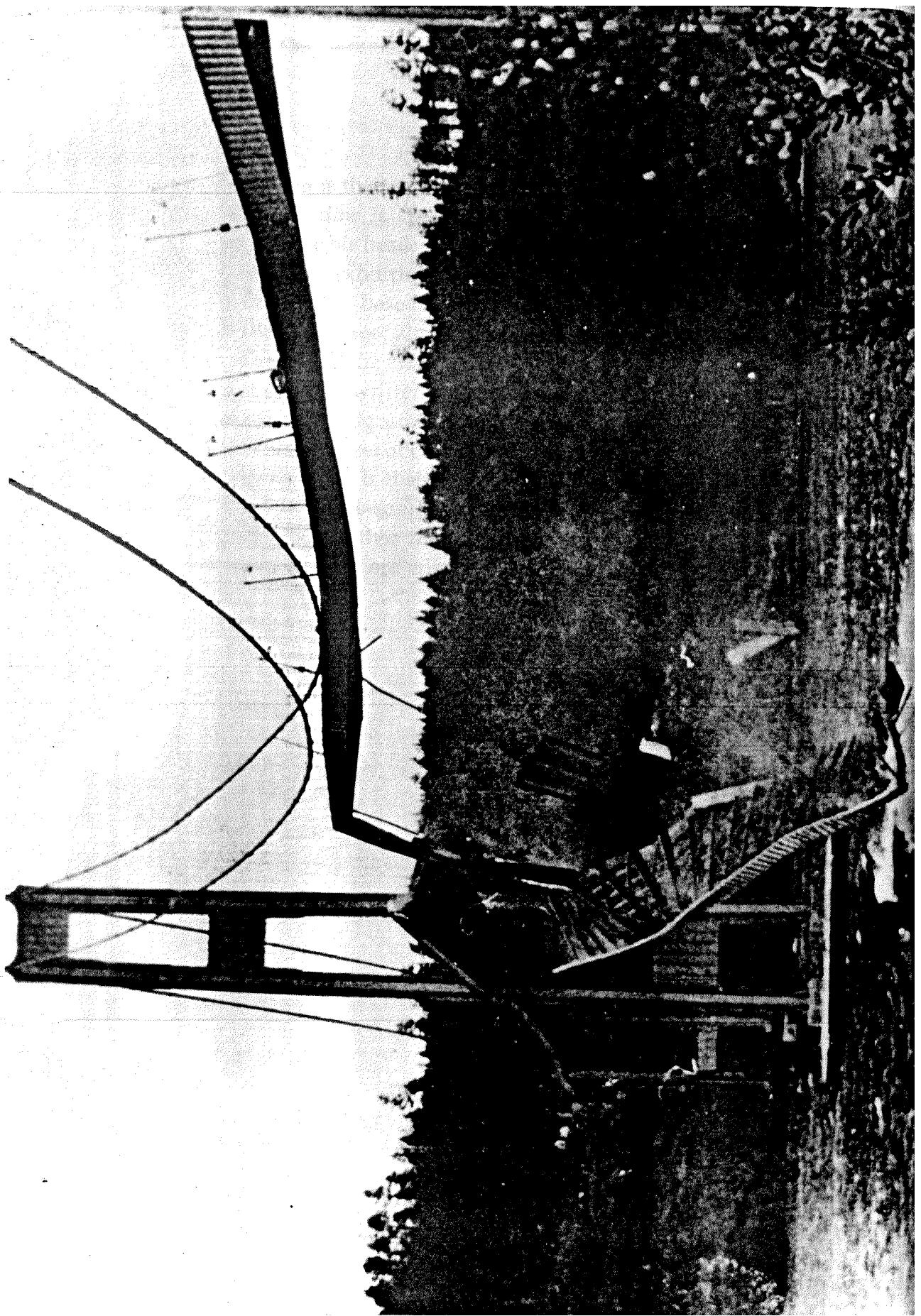
Um dos motivos é que apenas uma pequena parte - talvez 20% - do tempo do executivo é gasta em tarefas nas quais ele precisa de informações de fora de sua cabeça. O resto é comunicação: ouvir, relatar, ensinar, exortar, aconselhar, encorajar. Mas para a fração que é Com base em dados, o punhado de documentos essenciais é vital e atenderá a quase todas as necessidades.

A tarefa do gerente é desenvolver um plano e então realizá-lo. Mas apenas o plano escrito é preciso e comunicável. Esse plano consiste em documentos sobre o quê, quando, quanto, onde e quem. Esse pequeno conjunto de documentos essenciais engloba grande parte do trabalho do gerente. Se sua natureza abrangente e crítica for reconhecida no início, o gerente pode abordá-los como ferramentas amigáveis em vez de um trabalho incômodo irritante. Ele definirá sua direção com muito mais precisão e rapidez ao fazer isso.

onze

Plano de arremesso

Uma saída



onze

Plano de arremesso

Uma saída

Não há nada neste mundo constante, exceto inconstância.

RÁPIDO

*É bom senso pegar um método e experimentá-lo. Se falhar,
admita francamente e tente outro. Mas acima de tudo tente
something.*

FRANKLIN D. ROOSEVELT

Colapso da ponte Tacoma Narrows aerodinamicamente mal projetada,
1940

Foto UPI / Arquivo Bettman

Plantas Piloto e Ampliação

Os engenheiros químicos aprenderam há muito tempo que um processo que funciona em laboratório não pode ser implementado em uma fábrica em apenas uma etapa. Uma etapa intermediária chamada de *planta piloto* é necessário fornecer experiência em aumentar a escala de quantidades e operar em ambientes não protetores. Por exemplo, um processo de laboratório para dessalinização de água será testado em uma planta piloto com capacidade de 10.000 galões / dia antes de ser usado para um sistema comunitário de água de 2.000.000 galões / dia.

Os integradores de sistemas de programação também foram expostos a esta lição, mas parece que ainda não foi aprendida. Projeto após projeto projeta um conjunto de algoritmos e, em seguida, mergulha na construção de software para entrega ao cliente em um cronograma que exige a entrega da primeira coisa construída.

Na maioria dos projetos, o primeiro sistema construído quase não pode ser usado. Pode ser muito lento, muito grande, difícil de usar ou todos os três. Não há alternativa a não ser começar de novo, inteligente, porém mais inteligente, e construir uma versão redesenhada na qual esses problemas sejam resolvidos. O descarte e o redesenho podem ser feitos de uma só vez ou peça por peça. Mas toda experiência de grande sistema mostra que isso será feito.² Onde um novo conceito de sistema ou nova tecnologia é usado, é preciso construir um sistema para jogar fora, pois mesmo o melhor planejamento não é tão onisciente a ponto de acertar na primeira vez.

A questão da gestão, portanto, não é *se* para construir um sistema piloto e jogá-lo fora. Vocês *vai* faça isso. A única questão é *se* devemos planejar com antecedência a construção de um descartável ou se prometer entregá-lo aos clientes. Visto desta forma, a resposta é muito mais clara. Entregar esse lixo aos clientes ganha tempo, mas só ao custo da agonia para o usuário, distração para os construtores enquanto eles fazem o redesenho e uma má reputação para o produto que o melhor redesenho terá dificuldade em viver.

Portanto *planeje jogar um fora; você vai, de qualquer maneira.*

A única constância é a própria mudança

Uma vez que se reconhece que um sistema piloto deve ser construído e descartado, e que um redesenho com ideias alteradas é inevitável, torna-se útil enfrentar todo o fenômeno da mudança. O primeiro passo é aceitar o fato da mudança como um estilo de vida, em vez de uma exceção desagradável e irritante. Cosgrove apontou perceptivelmente que o programador fornece a satisfação da necessidade do usuário, em vez de qualquer produto tangível. E tanto a necessidade real quanto a percepção do usuário dessa necessidade mudarão conforme os programas são criados, testados e usados.³

É claro que isso também se aplica às necessidades atendidas por produtos de hardware, sejam carros novos ou novos computadores. Mas a própria existência de um objeto tangível serve para conter e quantizar a demanda do usuário por mudanças. Tanto a tratabilidade quanto a invisibilidade do produto de software expõem seus construtores a mudanças perpétuas nos requisitos.

Longe de mim sugerir que todas as mudanças nos objetivos e requisitos do cliente devem, podem ou devem ser incorporadas ao projeto. É claro que um limite deve ser estabelecido e deve ficar cada vez mais alto à medida que o desenvolvimento avança, ou nenhum produto jamais aparecerá.

No entanto, algumas mudanças de objetivos são inevitáveis e é melhor estar preparado para elas do que presumir que não virão. Mudanças no objetivo não são apenas inevitáveis, mas também na estratégia e na técnica de desenvolvimento. O conceito de jogar fora é em si apenas uma aceitação do fato de que, à medida que se aprende, ele muda o design.⁴

Planeje o sistema para a mudança

As formas de projetar um sistema para tal mudança são bem conhecidas e amplamente discutidas na literatura - talvez mais amplamente divulgada.

xingado do que praticado. Eles incluem modularização cuidadosa, sub-rotina extensa, definição precisa e completa de interfaces de intermodulos e documentação completa destes. Menos obviamente, alguém deseja sequências de chamada padrão e técnicas orientadas por tabela usadas sempre que possível.

O mais importante é o uso de uma linguagem de alto nível e técnicas de autodocumentação para reduzir os erros induzidos por mudanças. Usar operações de tempo de compilação para incorporar declarações padrão ajuda poderosamente a fazer alterações.

A quantização da mudança é uma técnica essencial. Cada produto deve ter versões numeradas e cada versão deve ter sua própria programação e uma data de congelamento, após a qual as alterações vão para a próxima versão.

Planeje a organização para a mudança

Cosgrove defende tratar todos os planos, marcos e cronogramas como provisórios, de modo a facilitar a mudança. Isso vai longe demais - a falha comum dos grupos de programação hoje é muito pouco controle de gerenciamento, não muito.

No entanto, ele oferece um grande insight. Ele observa que a relutância em documentar projetos não se deve apenas à preguiça ou à pressão do tempo. Em vez disso, vem da relutância do designer em se comprometer com a defesa de decisões que ele sabe serem provisórias. "Ao documentar um projeto, o designer se expõe às críticas de todos, e deve ser capaz de defender tudo o que escreve. Se a estrutura organizacional é ameaçadora de alguma forma, nada será documentado até que seja totalmente defensável".

Estruturar uma organização para mudanças é muito mais difícil do que projetar um sistema para mudanças. Cada homem deve ser designado para tarefas que o ampliem, de forma que toda a força seja tecnicamente flexível. Em um grande projeto, o gerente precisa manter dois ou três programadores de ponta como uma cavalaria técnica que pode galopar para o resgate onde quer que a batalha seja mais acirrada.

As estruturas de gerenciamento também precisam ser alteradas à medida que o sistema muda. Isso significa que o chefe deve dar muita atenção para manter seus gerentes e seu pessoal técnico tão intercambiáveis quanto seus talentos permitirem.

As barreiras são sociológicas e devem ser combatidas com vigilância constante. Em primeiro lugar, os próprios gerentes costumam pensar que os funcionários seniores são "valiosos demais" para serem usados na programação real. Em seguida, os cargos de gerenciamento têm maior prestígio. Para superar esse problema, alguns laboratórios, como o Bell Labs, abolem todos os cargos. Cada funcionário profissional é um "membro da equipe técnica". Outros, como a IBM, mantêm uma escada dupla de avanço, como mostra a Figura 11.1. Os degraus correspondentes são teoricamente equivalentes.

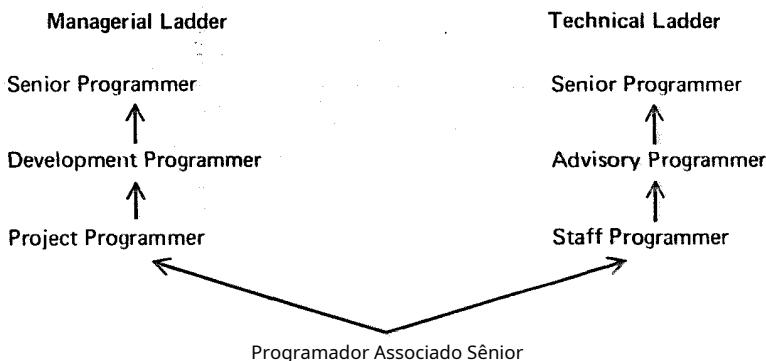


Fig. 11.1 Escada dupla de avanço da IBM

É fácil estabelecer escalas salariais correspondentes para os degraus. É muito mais difícil dar-lhes o prestígio correspondente. Os escritórios devem ter o mesmo tamanho e capacidade. Os serviços de secretariado e outros serviços de apoio devem corresponder. Uma transferência da escada técnica para um nível correspondente na hierarquia nunca deve ser acompanhada de um aumento, e deve ser anunciada sempre como

O problema fundamental com a manutenção do programa é que consertar um defeito tem uma chance substancial (20-50 por cento) de introduzir outro. Portanto, todo o processo envolve dois passos à frente e um passo atrás.

Por que os defeitos não são corrigidos de maneira mais limpa? Primeiro, mesmo um defeito sutil se mostra como uma falha local de algum tipo. Na verdade, muitas vezes tem ramificações em todo o sistema, geralmente não óbvias. Qualquer tentativa de consertá-lo com um esforço mínimo consertará o local e o óbvio, mas, a menos que a estrutura seja pura ou a documentação muito fina, os efeitos de longo alcance do reparo serão negligenciados. Em segundo lugar, o reparador geralmente não é o homem que escreveu o código e, muitas vezes, ele é um programador júnior ou estagiário.

Como consequência da introdução de novos bugs, a manutenção do programa requer muito mais testes de sistema por instrução escrita do que qualquer outra programação. Teoricamente, após cada correção, deve-se executar todo o banco de casos de teste anteriormente executados no sistema, para garantir que ele não tenha sido danificado de forma obscura. Na prática, tal *teste de regressão* deve de fato se aproximar desse ideal teórico, e é muito caro.

Claramente, os métodos de projetar programas de modo a eliminar ou pelo menos iluminar os efeitos colaterais podem ter uma grande recompensa em custos de manutenção. O mesmo pode acontecer com os métodos de implementação de projetos com menos pessoas, menos interfaces e, portanto, menos bugs.

Um passo à frente e um passo atrás

Lehman e Belady estudaram a história de lançamentos sucessivos em um grande sistema operacional.⁸ Eles descobriram que o número total de módulos aumenta linearmente com o número da versão, mas que o número de módulos afetados aumenta exponencialmente com o número da versão. Todos os reparos tendem a destruir a estrutura, a aumentar a entropia e a desordem do sistema. Cada vez menos esforço é gasto para consertar as falhas do projeto original; cada vez mais é gasto na correção de falhas introduzidas por correções anteriores. Com o passar do tempo, o sistema se torna cada vez menos organizado. Mais cedo ou mais tarde, o conserto

deixa de ganhar terreno. Cada passo para a frente é acompanhado por um para trás. Embora em princípio utilizável para sempre, o sistema se desgastou como base para o progresso. Além disso, as máquinas mudam, as configurações mudam e os requisitos do usuário mudam, de modo que o sistema não pode ser usado para sempre. É necessário um redesenho totalmente novo e de baixo para cima.

E assim, a partir de um modelo estatístico mecânico, Belady e Lehman chegam, para os sistemas de programação, a uma conclusão mais geral apoiada pela experiência de toda a Terra. "As coisas estão sempre no seu melhor no início", disse Pascal. CS Lewis afirmou isso de forma mais perspicaz:

Essa é a chave da história. Ótima energia é gasta - civilizações são construídas - excelentes instituições criadas; mas cada vez que algo dá errado. Alguma falha fatal sempre leva as pessoas egoísticas e cruéis ao topo, e então tudo volta à miséria e à ruína. Na verdade, a máquina trava. Parece que arranca bem, corre alguns metros e depois avaria. "¹

A construção do programa de sistemas é um processo de redução da entropia, portanto, inherentemente metaestável. A manutenção do programa é um processo que aumenta a entropia, e mesmo sua execução mais habilidosa apenas atrasa o afundamento do sistema em uma obsolescência não corrigível.

12

Ferramentas Sharp



12

Ferramentas Sharp

Um bom trabalhador é conhecido por suas ferramentas.

PROVÉRBIO

A. Pisano, "Lo Scultore", do Campanile di Santa Maria del Fiore, Florença, c. 1335
Scala / ArtResource, NY

Mesmo com essa data tardia, muitos projetos de programação ainda são operados como oficinas de máquinas no que diz respeito a ferramentas. Cada mestre mecânico tem seu próprio conjunto pessoal, coletado ao longo da vida e cuidadosamente trancado e guardado - as evidências visíveis de habilidades pessoais. Da mesma forma, o programador mantém pequenos editores, classificações, despejos binários, utilitários de espaço em disco, etc., armazenados em seu arquivo.

Essa abordagem, no entanto, é tola para um projeto de programação. Em primeiro lugar, o problema essencial é a comunicação, e as ferramentas individualizadas dificultam em vez de ajudar na comunicação. Em segundo lugar, a tecnologia muda quando alguém muda de máquina ou linguagem de trabalho, então o tempo de vida da ferramenta é curto. Finalmente, é obviamente muito mais eficiente ter um desenvolvimento e manutenção comuns das ferramentas de programação de uso geral.

No entanto, ferramentas de uso geral não são suficientes. Tanto as necessidades especializadas quanto as preferências pessoais determinam a necessidade de ferramentas especializadas também; portanto, ao discutir equipes de programação, postulei um criador de ferramentas por equipe. Este homem domina todas as ferramentas comuns e é capaz de instruir seu chefe-cliente sobre como usá-las. Ele também cria as ferramentas especializadas de que seu chefe precisa.

O gerente de um projeto, então, precisa estabelecer uma filosofia e reservar recursos para a construção de ferramentas comuns. Ao mesmo tempo, ele deve reconhecer a necessidade de ferramentas especializadas, e não invejar sua própria equipe de construção de ferramentas. Essa tentação é insidiosa. Sente-se que se todos os criadores de ferramentas espalhados fossem reunidos para aumentar a equipe de ferramentas comuns, resultaria em maior eficiência. Mas não é assim.

Quais são as ferramentas sobre as quais o gerente deve filosofar, planejar e organizar? Primeiro um *instalação de computador*. Isso requer máquinas e uma filosofia de programação deve ser adotada. Requer um *sistema operacional*, e filosofias de serviço devem ser estabelecidas. Isso requer *língua*, e uma política de linguagem deve ser estabelecida baixa. Então há *utilitários, auxiliares de depuração, geradores de casos de teste*, e um *sistema de processamento de texto* para lidar com a documentação. Vejamos um por um.¹

Máquinas Alvo

O suporte da máquina é usualmente dividido em *target machine* e a *máquinas de veículos*. A máquina de destino é aquela para a qual o software está sendo escrito e na qual deve ser testado. As máquinas veiculares são aquelas que prestam os serviços utilizados na construção do sistema. Se alguém está construindo um novo sistema operacional para uma máquina antiga, ele pode servir não apenas como alvo, mas também como veículo.

Que tipo de instalação-alvo? As equipes que criam novos supervisores ou outro software central do sistema, é claro, precisam de suas próprias máquinas. Esses sistemas precisarão de operadores e um ou dois programadores de sistema que mantenham o suporte padrão da máquina atualizado e em condições de uso.

Se uma máquina separada for necessária, é algo bastante peculiar - não precisa ser rápido, mas precisa de pelo menos um milhão de bytes de armazenamento principal, cem milhões de bytes de disco on-line e terminais. Apenas terminais alfanuméricos são necessários, mas eles devem ir muito mais rápido do que os 15 caracteres por segundo que caracterizam as máquinas de escrever. Uma grande memória aumenta muito a produtividade, permitindo que a sobreposição e o corte de tamanho sejam feitos após o teste funcional.

A máquina de depuração, ou seu software, também precisa ser instrumentada, de modo que contagens e medições de todos os tipos de parâmetros do programa possam ser feitas automaticamente durante a depuração. Os padrões de uso de memória, por exemplo, são diagnósticos poderosos das causas de comportamento lógico estranho ou desempenho inesperadamente lento.

Agendamento. Quando a máquina de destino está correta, como quando seu primeiro sistema operacional está sendo construído, o tempo da máquina é escasso e sua programação é um grande problema. A necessidade de tempo de máquina-alvo tem uma curva de crescimento peculiar. No desenvolvimento do OS / 360 tivemos bons simuladores System / 360 e outros veículos. Com a experiência anterior, projetamos quantas horas de tempo S / 360 precisaríamos e começamos a adquirir as primeiras máquinas de produção da fábrica

ção. Mas eles ficavam parados, mês após mês. Então, de repente, todos os 16 sistemas estavam totalmente carregados, e o problema era o racionamento. A utilização parecia algo como a Fig. 12.1. Todos começaram a depurar seus primeiros componentes ao mesmo tempo e, a partir daí, a maioria da equipe estava constantemente depurando algo.

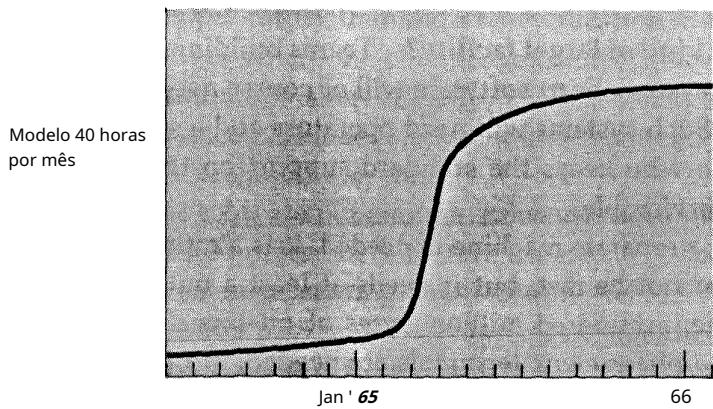


Fig. 12.1 Crescimento no uso de máquinas de destino

Centralizamos todas as nossas máquinas e biblioteca de fitas e montamos uma equipe profissional e experiente na sala de máquinas para operá-las. Para maximizar o tempo escasso do S / 360, executamos todas as execuções de depuração em lote em qualquer sistema que fosse gratuito e apropriado. Tentamos quatro doses por dia (tempo de retorno de duas horas e meia) e exigimos tempo de retorno de quatro horas. Um auxiliar 1401 com terminais foi usado para agendar execuções, para controlar os milhares de trabalhos e para monitorar o tempo de retorno.

Mas toda aquela organização estava exagerada. Depois de alguns meses de recuperação lenta, recriminações mútuas e outras agonias, passamos a alocar o tempo da máquina em blocos substanciais.

Toda a equipe de classificação de quinze homens, por exemplo, receberia um sistema para um bloco de quatro a seis horas. Cabia a eles se programarem. Se ficasse ocioso, nenhum estranho poderia usá-lo.

Isso, ele desenvolve, era a melhor maneira de alocar e programar. Embora a utilização da máquina possa ter sido um pouco menor (e geralmente não era), a produtividade aumentou. Para cada homem nessa equipe, dez arremessos em um bloco de seis horas são muito mais produtivos do que dez arremessos com intervalos de três horas, porque a concentração sustentada reduz o tempo de raciocínio. Depois de tal sprint, uma equipe geralmente precisava de um ou dois dias para colocar em dia a papelada antes de solicitar outro bloqueio.

Freqüentemente, apenas três programadores podem compartilhar e programar um bloco de tempo de maneira proveitosa. Esta parece ser a melhor maneira de usar uma máquina de destino ao depurar um novo sistema operacional.

Sempre foi assim na prática, embora nunca na teoria. A depuração do sistema sempre foi uma ocupação do turno da noite, como a astronomia. Vinte anos atrás, na década de 70, fui iniciado na informalidade produtiva da madrugada, quando todos os chefes da sala de máquinas estão dormindo profundamente em casa e os operadores não gostam de ser rigorosos nas regras. Três gerações de máquinas se passaram; as tecnologias mudaram totalmente; sistemas operacionais surgiram; e, no entanto, este método preferido de trabalho não mudou. Ele perdura porque é muito produtivo. Chegou a hora de reconhecer sua produtividade e abraçar a prática frutífera abertamente.

Máquinas de veículos e serviços de dados

Simuladores. Se o computador de destino for novo, é necessário um simulador lógico para ele. Isso fornece um veículo de depuração muito antes de o destino real existir. Tão importante quanto, dá acesso a um *confiável* veículo de depuração mesmo depois de ter uma máquina de destino disponível.

Confiável não é o mesmo que *preciso*. O simulador certamente falhará em algum aspecto em ser um implemento fiel e preciso

da arquitetura da nova máquina. Mas será o *mesmo* implementação de um dia para o outro, e o novo hardware não.

Hoje em dia, estamos acostumados a ter o hardware do computador funcionando corretamente quase o tempo todo. A menos que um programador de aplicativo veja um sistema se comportando de maneira inconsistente de execução para execução idêntica, é aconselhável procurar por bugs em seu código, em vez de em seu mecanismo.

Essa experiência, porém, é um péssimo treinamento para a programação de suporte de uma nova máquina. Construído em laboratório, pré-produção ou hardware inicial, *não* trabalha como definido, faz *não* trabalha de forma confiável e faz *não* permanecer o mesmo dia a dia. Conforme os bugs são encontrados, as alterações de engenharia são feitas em todas as cópias da máquina, incluindo aquelas do grupo de programação. Essa base móvel já é ruim o suficiente. As falhas de hardware, geralmente intermitentes, são piores. A incerteza é a pior de tudo, pois rouba a pessoa do incentivo para cavar diligentemente em seu código em busca de um bug - ele pode nem estar lá. Portanto, um simulador confiável em um veículo bem envelhecido mantém sua utilidade por muito mais tempo do que seria de se esperar.

Veículos compiladores e montadores. Pelas mesmas razões, um deseja compiladores e montadores que rodam em veículos confiáveis, mas compilam código-objeto para o sistema de destino. Isso pode então começar a ser depurado no simulador.

Com a programação de linguagem de alto nível, pode-se fazer grande parte da depuração compilando e testando o código-objeto na máquina do veículo antes de começar a testar o código da máquina-alvo. Isso dá a eficiência da execução direta, ao invés da simulação, combinada com a confiabilidade da máquina estável.

Bibliotecas de programas e contabilidade. Um muito bem sucedido e importante uso de uma máquina de veículo no esforço de desenvolvimento do OS / 360 foi para a manutenção de bibliotecas de programas. Um sistema desenvolvido sob a liderança de WR Crowley tinha dois 7010's conectados, compartilhando um grande banco de dados em disco. O 7010 também forneceu um S / 360

assembler. Todo o código testado ou em teste foi mantido nesta biblioteca, tanto o código-fonte quanto os módulos de carregamento montados. A biblioteca foi de fato dividida em sub-bibliotecas com diferentes regras de acesso.

Primeiro, cada grupo ou programador tinha uma área onde mantinha cópias de seus programas, seus casos de teste e a estrutura de que precisava para os testes de componentes. Nisso *cercadinho* área não havia restrições sobre o que um homem poderia fazer com seus próprios programas; eles eram dele.

Quando um homem tinha seu componente pronto para integração em uma parte maior, ele passava uma cópia para o gerente desse sistema maior, que a colocava em um *integração de sistema library*. Agora o programador original não poderia alterá-lo, exceto com a permissão do gerenciador de integração. À medida que o sistema era montado, o último continuava com todos os tipos de testes de sistema, identificando bugs e obtendo correções.

De vez em quando, uma versão do sistema fica pronta para uso mais amplo. Em seguida, seria promovido para o *sub-biblioteca da versão atual*. Esta cópia era sacrossanta, tocada apenas para consertar bugs incapacitantes. Ele estava disponível para uso na integração e teste de todas as novas versões do módulo. Um diretório de programa no 7010 acompanhava cada versão de cada módulo, seu status, localização e mudanças.

Duas noções são importantes aqui. O primeiro é *ao controle*, a ideia de cópias do programa pertencentes a gerentes que são os únicos que podem autorizar sua mudança. O "segundo é o de *separação formal e progressão* do cercadinho, para a integração, para o lançamento.

Na minha opinião, essa foi uma das coisas mais bem feitas no esforço do OS / 360. É uma tecnologia de gerenciamento que parece ter sido desenvolvida de forma independente em vários projetos de programação massivos, incluindo os do Bell Labs, ICL e da Universidade de Cambridge.⁸ É aplicável tanto à documentação quanto aos programas. É uma tecnologia indispensável.

Ferramentas do programa. À medida que novas técnicas de depuração aparecem, as antigas diminuem, mas não desaparecem. Portanto, são necessários despejos, editores de arquivos de origem, despejos de instantâneos e até mesmo rastreios.

Da mesma forma, é necessário um conjunto completo de utilitários para colocar decks em discos, fazer cópias em fitas, imprimir arquivos, alterar catálogos. Se alguém contratar um criador de ferramentas de projeto no início do processo, isso pode ser feito uma vez e pode estar pronto no momento em que forem necessários.

Sistema de documentação. Entre todas as ferramentas, a que economiza mais trabalho pode muito bem ser um sistema de edição de texto computadorizado, operando em um veículo confiável. Tínhamos um muito útil, criado por J. W. Franklin. Sem isso, espero que o manual do OS / 360 tenha sido muito posterior e mais enigmático. Há quem argumente que a prateleira de manuais do OS / 360 com mais de um metro de altura representa diarreia verbal, que a própria voluminosidade introduz um novo tipo de incompreensibilidade. E há alguma verdade nisso.

Mas eu respondo de duas maneiras. Primeiro, a documentação do OS / 360 é avassaladora em massa, mas o plano de leitura é cuidadosamente delineado; se alguém o usa seletivamente, pode ignorar a maior parte do tempo. Deve-se considerar a documentação do OS / 360 como uma biblioteca ou enciclopédia, não um conjunto de textos obrigatórios.

Em segundo lugar, isso é muito preferível à severa subdocumentação que caracteriza a maioria dos sistemas de programação. Eu concordarei rapidamente, entretanto, que a redação poderia ser amplamente melhorada em alguns lugares, e que o resultado de uma redação melhor seria reduzido em volume. Algumas partes (por exemplo, *Conceitos e Instalações*) estão muito bem escritos agora.

Simulador de desempenho. Melhor ter um. Construa-o de fora para dentro, como discutiremos no próximo capítulo. Use o mesmo design de cima para baixo para o simulador de desempenho, o simulador lógico e o produto. Comece bem cedo. Ouça quando ele fala.

Linguagem de alto nível e programação interativa

As duas ferramentas mais importantes para a programação do sistema hoje são duas que não eram usadas no desenvolvimento do OS / 360 há quase uma década. Eles ainda não são amplamente usados, mas todas as evidências apontam para seu poder e aplicabilidade. Eles são (1) linguagem de alto nível e (2) programação interativa. Estou convencido de que apenas a inércia e

a preguiça impede a adoção universal dessas ferramentas; as dificuldades técnicas não são mais desculpas válidas.

Linguagem de alto nível. Os principais motivos para usar uma linguagem de alto nível são produtividade e velocidade de depuração. Discutimos a produtividade anteriormente (Capítulo 8). Não há muitas evidências numéricas, mas o que há sugere melhoria por fatores integrais, não apenas percentuais incrementais.

A melhoria da depuração vem do fato de que há menos bugs e são mais fáceis de encontrar. Há menos porque evita-se todo um nível de exposição ao erro, um nível no qual cometemos não apenas erros sintáticos, mas semânticos, como uso indevido de registros. Os bugs são mais fáceis de encontrar porque os diagnósticos do compilador ajudam a localizá-los e, mais importante, porque é muito fácil inserir instantâneos de depuração.

Para mim, esses motivos de produtividade e depuração são esmagadores. Não consigo conceber facilmente um sistema de programação *eu* seria construído em linguagem assembly.

Bem, e quanto às objeções clássicas a tal ferramenta? São três: não me deixa fazer o que quero. O código do objeto é muito grande. O código do objeto é muito lento.

Quanto à função, acredito que a objeção não é mais válida. Todo testemunho indica que uma pessoa pode fazer o que precisa, mas dá trabalho descobrir como, e ocasionalmente pode precisar de artifícios desagradáveis.^{3/4}

Quanto ao espaço, os novos compiladores de otimização estão começando a ser muito satisfatórios e esse aprimoramento continuará.

Quanto à velocidade, a otimização de compiladores agora produz algum código que é mais rápido do que o código escrito à mão do programador. Além disso, normalmente é possível resolver problemas de velocidade substituindo de um a cinco por cento de um programa gerado pelo compilador por um substituto escrito à mão após o primeiro ser totalmente depurado.⁵

Que linguagem de alto nível deve-se usar para a programação do sistema? O único candidato razoável hoje é PL / I.⁸ Tem muito

conjunto completo de funções; é compatível com os ambientes do sistema operacional; e uma variedade de compiladores estão disponíveis, alguns interativos, alguns rápidos, alguns muito diagnósticos e alguns produzindo código altamente otimizado. Eu mesmo acho mais rápido desenvolver algoritmos em APL; em seguida, traduzo para PL / I para corresponder ao ambiente do sistema.

Programação interativa. Uma das justificativas para o projeto Multics do MIT foi sua utilidade para a construção de sistemas de programação. Multics (e depois disso, o TSS da IBM) difere em conceito de outros sistemas de computação interativos exatamente naqueles aspectos necessários para a programação de sistemas: muitos níveis de compartilhamento e proteção de dados e programas, gerenciamento extensivo de biblioteca e recursos para trabalho cooperativo entre usuários de terminais. Estou convencido de que os sistemas interativos nunca substituirão os sistemas em lote para muitas aplicações. Mas acho que a equipe do Multics apresentou seu caso mais convincente no aplicativo de programação do sistema.

Ainda não há muitas evidências disponíveis sobre a verdadeira fecundidade de tais ferramentas aparentemente poderosas. Lá é *um* reconhecimento generalizado de que a depuração é a parte difícil e lenta da programação do sistema, e o retorno lento é a ruína da depuração. Portanto, a lógica da programação interativa parece inexorável.⁷

Program	Size	Batch (B) or Conversational (C)	Instructions/man-year
ESS code	800,000	B	500-1000
7094 ESS support	120,000	B	2100-3400
360 ESS support	32,000	C	8000
360 ESS support	8,300	B	4000

Fig. 12.2 Produtividade comparativa em lote e conversacional programática

Além disso, ouvimos bons testemunhos de muitos que construíram pequenos sistemas ou partes de sistemas dessa maneira. Os únicos números que vi para efeitos na programação de grandes sistemas foram relatados por John Harr da Bell Labs. Eles são mostrados na Fig. 12.2. Esses números são para escrever, montar e depurar programas. O primeiro programa é basicamente um programa de controle; os outros três são tradutores, editores e outros. Os dados da Hair sugerem que um recurso interativo pelo menos dobra a produtividade na programação do sistema.⁸

O uso eficaz da maioria das ferramentas interativas requer que o trabalho seja feito em uma linguagem de alto nível, pois os terminais de teletipo e máquina de escrever não podem ser usados para depurar despejando memória. Com uma linguagem de alto nível, a fonte pode ser facilmente editada e as impressões seletivas feitas facilmente. Juntos, eles realmente fazem um par de ferramentas afiadas.

13

O todo e as partes



13

O todo e as partes

Posso invocar espíritos do vasto fundo.

Por que isso pode /, ou qualquer homem pode; mas eles virão quando você os chamar?

SHAKESPEARE. KING HENRY W, PARTI

A magia moderna, como a antiga, tem seus praticantes orgulhosos: "Posso escrever programas que controlam o tráfego aéreo, interceptam mísseis balísticos, reconciliam contas bancárias, controlam linhas de produção." Ao que vem a resposta: "Eu também posso, e também qualquer homem, mas eles funcionam quando você os escreve?"

Como alguém constrói um programa para funcionar? Como se testa um programa? E como alguém integra um conjunto testado de programas de componentes em um sistema testado e confiável? Já abordamos as técnicas aqui e ali; vamos agora considerá-los um pouco mais sistematicamente.

Projetando theBugsOut

A definição à prova de bugs. Os bugs mais perniciosos e sutis são bugs de sistema que surgem de suposições incompatíveis feitas pelos autores de vários componentes. A abordagem à integridade conceitual discutida acima nos Capítulos 4, 5 e 6 aborda esses problemas diretamente. Resumindo, a integridade conceitual do produto não apenas o torna mais fácil de usar, mas também torna mais fácil de construir e menos sujeito a bugs.

O mesmo acontece com o esforço arquitetônico detalhado e meticuloso implícito nessa abordagem. VA Vyssotsky, do Projeto Safeguard da Bell Telephone Laboratories, diz: "A tarefa crucial é definir o produto. Muitas, muitas falhas dizem respeito exatamente àqueles aspectos que nunca foram totalmente especificados."¹ Definição cuidadosa de função, especificação cuidadosa e o exorcismo disciplinado de babados de função e voos de técnica reduzem o número de bugs de sistema que precisam ser encontrados.

Testando a especificação. Muito antes de qualquer código existir, a especificação deve ser entregue a um grupo de teste externo para ser examinada quanto à integridade e clareza. Como diz Vyssotsky, os próprios desenvolvedores não podem fazer isso: "Eles não vão te dizer que não entendem; eles vão inventar o caminho através das lacunas e obscuridades."

Design de cima para baixo. Em um artigo muito claro de 1971, Niklaus Wirth formalizou um procedimento de design que havia sido usado durante anos pelos melhores programadores.² Além disso, suas noções, embora declaradas para o design de programas, aplicam-se completamente ao design de sistemas complexos de programas. A divisão da construção do sistema em arquitetura, implementação e realização é uma personificação dessas noções; além disso, cada arquitetura, implementação e realização pode ser melhor realizada por métodos de cima para baixo.

Resumidamente, o procedimento de Wirth é identificar o design como uma sequência de *etapas de refinamento*. Um esboça uma definição de tarefa grosseira e um método de solução grosseira que atinge o resultado principal. Em seguida, examina-se a definição mais de perto para ver como o resultado difere do que é desejado e dá-se as grandes etapas da solução e as divide em etapas menores. Cada refinamento na definição da tarefa torna-se um refinamento no algoritmo para solução, e cada um pode ser acompanhado por um refinamento na representação dos dados.

A partir deste processo, identifica-se *módulos* de solução ou de dados cujo refinamento posterior pode ocorrer independentemente de outro trabalho. O grau desta modularidade determina a adaptabilidade e mutabilidade do programa.

Wirth defende o uso de uma notação de alto nível possível em cada etapa, expondo os conceitos e ocultando os detalhes até que um refinamento posterior se torne necessário.

Um bom design de cima para baixo evita bugs de várias maneiras. Primeiro, a clareza da estrutura e representação torna mais fácil a declaração precisa dos requisitos e funções dos módulos. Em segundo lugar, o particionamento e a independência dos módulos evita bugs no sistema. Terceiro, a supressão de detalhes torna as falhas na estrutura mais aparentes. Quarto, o design pode ser testado em cada uma de suas etapas de refinamento, de modo que o teste pode começar mais cedo e se concentrar no nível de detalhe adequado em cada etapa.

O processo de refinamento gradual não significa que nunca se tenha que voltar, descartar o nível superior e começar tudo

novamente ao encontrar algum detalhe inesperadamente complicado. Na verdade, isso acontece com frequência. Mas é muito mais fácil ver exatamente quando e por que se deve jogar fora um design bruto e começar de novo. Muitos sistemas ruins vêm de uma tentativa de salvar um projeto básico ruim e remendá-lo com todos os tipos de alívio cosmético. O design de cima para baixo reduz a tentação.

Estou convencido de que o design de cima para baixo é a nova formalização de programação mais importante da década.

Programação estruturada. Outro conjunto importante de novas ideias para projetar os bugs de programas deriva em grande parte de Dijkstra,³ e é construído sobre uma estrutura teórica de Bohm e Jacopini.⁴

Basicamente, a abordagem é projetar programas cujas estruturas de controle consistem apenas em loops definidos por uma instrução como DO WHILE e porções condicionais delineadas em grupos de instruções marcadas com colchetes e condicionadas por um IF ... THEN ... OUTRO. Bohm e Jacopini mostram que essas estruturas são teóricas suficientemente suficiente; Dijkstra argumenta que a alternativa, ramificação irrestrita via GO TO, produz estruturas que se prestam a erros lógicos.

A noção básica certamente é sólida. Muitas críticas foram feitas, e estruturas de controle adicionais, como um ramal em duas direções (a chamada instrução CASE) para distinguir entre muitas contingências, e um resgate de desastre (GO TO ANORMAL END) são muito convenientes. Além disso, alguns se tornaram muito doutrinários sobre como evitar todos os GO TO, e isso parece excessivo.

O ponto importante, e vital para a construção de programas livres de bugs, é que se deseja pensar nas estruturas de controle de um sistema como estruturas de controle, não como instruções de ramificação individuais. Essa forma de pensar é um grande passo em frente.

Depuração de componentes

Os procedimentos para depurar programas passaram por um grande ciclo nos últimos vinte anos e, de certa forma, eles estão de volta

onde eles começaram. O ciclo passou por quatro etapas e é divertido rastreá-los e ver a motivação de cada um.

Depuração na máquina. As primeiras máquinas tinham equipamento de entrada-saída relativamente pobre e longos atrasos de entrada-saída. Normalmente, a máquina lia e escrevia fita de papel ou fita magnética e instalações off-line eram usadas para preparação e impressão de fita. Isso tornou a entrada-saída da fita insuportavelmente inadequada para depuração, então o console foi usado em seu lugar. Assim, a depuração foi projetada para permitir tantas tentativas quanto possível por sessão da máquina.

O programador projetou cuidadosamente seu procedimento de depuração - planejar onde parar, quais locais de memória examinar, o que encontrar lá e o que fazer se não o fez. Essa programação meticulosa de si mesmo como uma máquina de depuração pode muito bem levar a metade do tempo de escrever o programa de computador a ser depurado.

O pecado principal foi empurrar o START com ousadia sem ter segmentado o programa em seções de teste com paradas planejadas.

Memorydumps. A depuração na máquina foi muito eficaz. Em uma sessão de duas horas, pode-se obter talvez uma dúzia de fotos. Mas os computadores eram muito escassos e muito caros, e o pensar em todo aquele tempo de máquina sendo desperdiçado era horrível.

Portanto, quando as impressoras de alta velocidade foram conectadas online, a técnica mudou. Um executou um programa até que uma verificação falhou e, em seguida, despejou toda a memória. Em seguida, começou o laborioso trabalho de mesa, contabilizando o conteúdo de cada local de memória. O tempo de mesa não era muito diferente daquele da depuração na máquina; mas ocorreu depois da execução do teste, na decifração, e não antes, no planejamento. A depuração para qualquer usuário em particular demorava muito mais, porque os disparos de teste dependiam do tempo de resposta do lote. Todo o procedimento, entretanto, foi projetado para minimizar o uso do tempo do computador e servir ao maior número possível de programadores.

Instantâneos. As máquinas nas quais o despejo de memória foi desenvolvido tinham de 2.000 a 4.000 palavras, ou 8 K a 16 Kbytes de memória. Mas os tamanhos de memória aumentaram aos trancos e barrancos, e o despejo total de memória tornou-se impraticável. Sopeople desenvolveu técnicas para seleção

despejo ativo, rastreamento seletivo e para inserir instantâneos em programas. O OS / 360 TESTRAN é o fim da linha nessa direção, permitindo inserir instantâneos em um programa sem remontagem ou recompilação.

Depuração interativa. Em 1959 Codd e seus colegas de trabalho⁵ e Strachey⁶ cada trabalho relatado tem como objetivo a depuração compartilhada por tempo, uma forma de obter tanto o retorno instantâneo da depuração na máquina quanto o uso eficiente da depuração em lote por máquina. O computador teria vários programas na memória, prontos para execução. Um terminal, controlado apenas pelo programa, seria associado a cada programa sendo depurado. A depuração estaria sob o controle de um programa de supervisão. Quando o programador em um terminal parava seu programa para examinar o progresso ou para fazer alterações, o supervisor executava outro programa, mantendo assim as máquinas ocupadas.

O sistema de multiprogramação de Codd foi desenvolvido, mas a ênfase estava no aprimoramento da taxa de transferência por meio da utilização eficiente de entrada-saída, e a depuração interativa não foi implementada. As ideias de Strachey foram aprimoradas e implementadas em 1963 em um sistema experimental para o 7090 por Corbato e colegas do MIT.⁷

Este desenvolvimento levou ao MULTICS, TSS e outros sistemas de compartilhamento de tempo de hoje.

As principais diferenças percebidas pelo usuário entre a depuração na máquina como praticada inicialmente e a depuração interativa de hoje são as facilidades possibilitadas pela presença da programação supervisora e seus intérpretes de linguagem associados. Pode-se programar e depurar em uma linguagem de alto nível. Os eficientes recursos de edição facilitam as alterações e os instantâneos.

O retorno à capacidade de retorno instantâneo da depuração na máquina ainda não trouxe um retorno ao planejamento prévio das sessões de depuração. Em certo sentido, esse planejamento prévio não é tão necessário como antes, uma vez que o tempo da máquina não se esgota enquanto alguém pensa.

No entanto, os resultados experimentais interessantes de Gold mostram que três vezes mais progresso na depuração interativa é feito na primeira interação de cada sessão do que na interação subsequente.

ções.⁸ Isso sugere fortemente que não estamos percebendo o potencial da interação devido à falta de planejamento da sessão. Chegou a hora de tirar o pó das velhas técnicas na máquina.

Acho que o uso adequado de um bom sistema de terminal requer duas horas na mesa para cada sessão de duas horas no terminal. Metade desse tempo é gasto em varredura após a última sessão: atualizando meu log de depuração, arquivando listagens de programas atualizadas em meu bloco de notas do sistema, explicando fenômenos estranhos. A outra metade é gasta na preparação: planejando mudanças e melhorias e projetando testes detalhados para a próxima vez. Sem esse planejamento, é difícil permanecer produtivo por até duas horas. Sem a varredura pós-sessão, é difícil manter a sucessão de sessões terminais sistemática e em movimento para a frente.

Casos de teste. Quanto ao design de procedimentos reais de depuração e casos de teste, Gruenberger tem um tratamento especialmente bom,⁹ e há tratamentos mais curtos em outros textos padrão.¹⁰

Depuração do sistema

A parte inesperadamente difícil de construir um sistema de programação é o teste do sistema. Já discuti algumas das razões para a dificuldade e seu caráter inesperado. De tudo isso, devemos nos convencer de duas coisas: a depuração do sistema levará mais tempo do que o esperado e sua dificuldade justifica uma abordagem totalmente sistemática e planejada. Vejamos agora o que envolve tal abordagem.¹²

Use componentes depurados. O bom senso, se não a prática comum, dita que se deve iniciar a depuração do sistema apenas depois que as peças parecem funcionar.

A prática comum se afasta disso de duas maneiras. Em primeiro lugar, é a abordagem do parafuso-it-together-and-try. Isso parece ser baseado na noção de que haverá bugs de sistema (ou seja, interface) além dos bugs de componentes. Quanto mais cedo alguém colocar as peças juntas, mais cedo surgirão os bugs do sistema. Um pouco menos sofisticada é a noção de que, usando as peças para testar umas às outras, um

evita muitos andaimes de teste. Ambos são obviamente verdadeiros, mas a experiência mostra que eles não são toda a verdade - o uso de componentes limpos e depurados economiza muito mais tempo em testes de sistema do que o gasto em andaimes e testes completos de componentes.

Um pouco mais sutil é a abordagem do "bug documentado". Isso diz que um componente está pronto para entrar no teste do sistema quando todas as falhas são *encontrado*, bem antes do tempo em que todos são *fixo*. Então, no teste de sistema, continua a teoria, a pessoa conhece os efeitos esperados desses bugs e pode ignorar esses efeitos, concentrando-se nos novos fenômenos.

Tudo isso é apenas um pensamento positivo, inventado para racionalizar a dor de horários desajustados. Um faz *não* conheça todos os efeitos esperados de bugs conhecidos. Se as coisas fossem simples, o teste do sistema não seria difícil. Além disso, a correção dos bugs de componentes documentados certamente injetará bugs desconhecidos e, então, o teste do sistema será confuso.

Construa muitos andaimes. Por andaime, quero dizer todos os programas e dados construídos para fins de depuração, mas nunca com a intenção de estar no produto final. Não é irracional que haja metade do código no scaffolding do que no produto.

Uma forma de andaime é o *componente fictício*, que consiste apenas em interfaces e talvez alguns dados falsos ou alguns pequenos casos de teste. Por exemplo, um sistema pode incluir um programa de classificação que ainda não foi concluído. Seus vizinhos podem ser testados usando um programa fictício que apenas lê e testa o formato dos dados de entrada e emite um conjunto de dados bem formatados, sem sentido, mas ordenados.

Outra forma é o *arquivo em miniatura*. Uma forma muito comum de bug do sistema é a má compreensão dos formatos de arquivos de fita e disco. Portanto, vale a pena construir alguns pequenos arquivos que tenham apenas alguns registros típicos, mas todas as descrições, ponteiros, etc.

O caso limite do arquivo em miniatura é o *Arquivo fictício*, que realmente não existe. O Job Control Language do OS / 360 fornece essa facilidade e é extremamente útil para depuração de componentes.

Ainda outra forma de andaime são *programas auxiliares*. Geradores para dados de teste, impressões de análises especiais, analisadores de tabela de referência cruzada são todos exemplos de gabaritos e acessórios para fins especiais que se pode querer construir.¹³

Mudanças de controle. O controle rígido durante o teste é uma das técnicas impressionantes de depuração de hardware e também se aplica a sistemas de software.

Primeiro, alguém deve estar no comando. Ele e somente ele deve autorizar alterações de componentes ou substituição de uma versão por outra.

Então, como discutido acima, deve haver cópias controladas do sistema: uma cópia bloqueada das versões mais recentes, usada para teste de componentes; uma cópia em teste, com correções sendo instaladas; cópias de cercadinho onde cada homem pode trabalhar em seu componente, fazendo correções e extensões.

Nos modelos de engenharia System / 360, viam-se fios ocasionais de fio roxo entre os fios amarelos de rotina. Quando um bug foi encontrado, duas coisas foram feitas. A correção Aquick foi concebida e instalada no sistema, para que os testes pudessem prosseguir. Essa mudança foi colocada em um fio roxo, de modo que ficou para fora como um polegar machucado. Foi inserido no log. Enquanto isso, um documento oficial de mudança foi preparado e iniciado na fábrica de automação de design. Eventualmente, isso resultou em desenhos atualizados e listas de fios, e um novo painel traseiro no qual a mudança foi implementada no circuito impresso ou fio amarelo. Agora o modelo físico e o papel estavam juntos novamente, e o fio roxo havia sumido.

A programação precisa de uma técnica de fio roxo e precisa desesperadamente de um controle rígido e profundo respeito pelo papel que, em última análise, é o produto. Os ingredientes vitais de tal técnica são o registro de todas as alterações em um diário e a distinção, realizada de forma visível no código-fonte, entre patches rápidos e correções bem pensadas, testadas e documentadas.

Adicione um componente de cada vez. Esse preceito também é óbvio, mas o otimismo e a preguiça nos tentam a violá-lo. Para fazer isso requer

150 O todo e as partes

manequins e outros andaimes, e isso dá trabalho. E, afinal, talvez todo esse trabalho não seja necessário? Talvez não haja bugs?

Não! Resistir a tentação! É disso que se trata o teste sistemático de sistema. Deve-se presumir que haverá muitos bugs e planejar um procedimento ordenado para eliminá-los.

Observe que é necessário ter casos de teste completos, testando os sistemas parciais após cada nova peça ser adicionada. E os antigos, executados com sucesso na última soma parcial, devem ser executados novamente no novo para testar a regressão do sistema.

Quantize atualizações. À medida que o sistema surge, os construtores de componentes aparecem de tempos em tempos, trazendo novas versões de suas peças - mais rápido, menor, mais completo ou, supostamente, menos cheio de erros. A substituição de um componente funcional por uma nova versão requer o mesmo procedimento de teste sistemático que a adição de um novo componente requer, embora deva exigir menos tempo, pois casos de teste mais completos e eficientes geralmente estarão disponíveis.

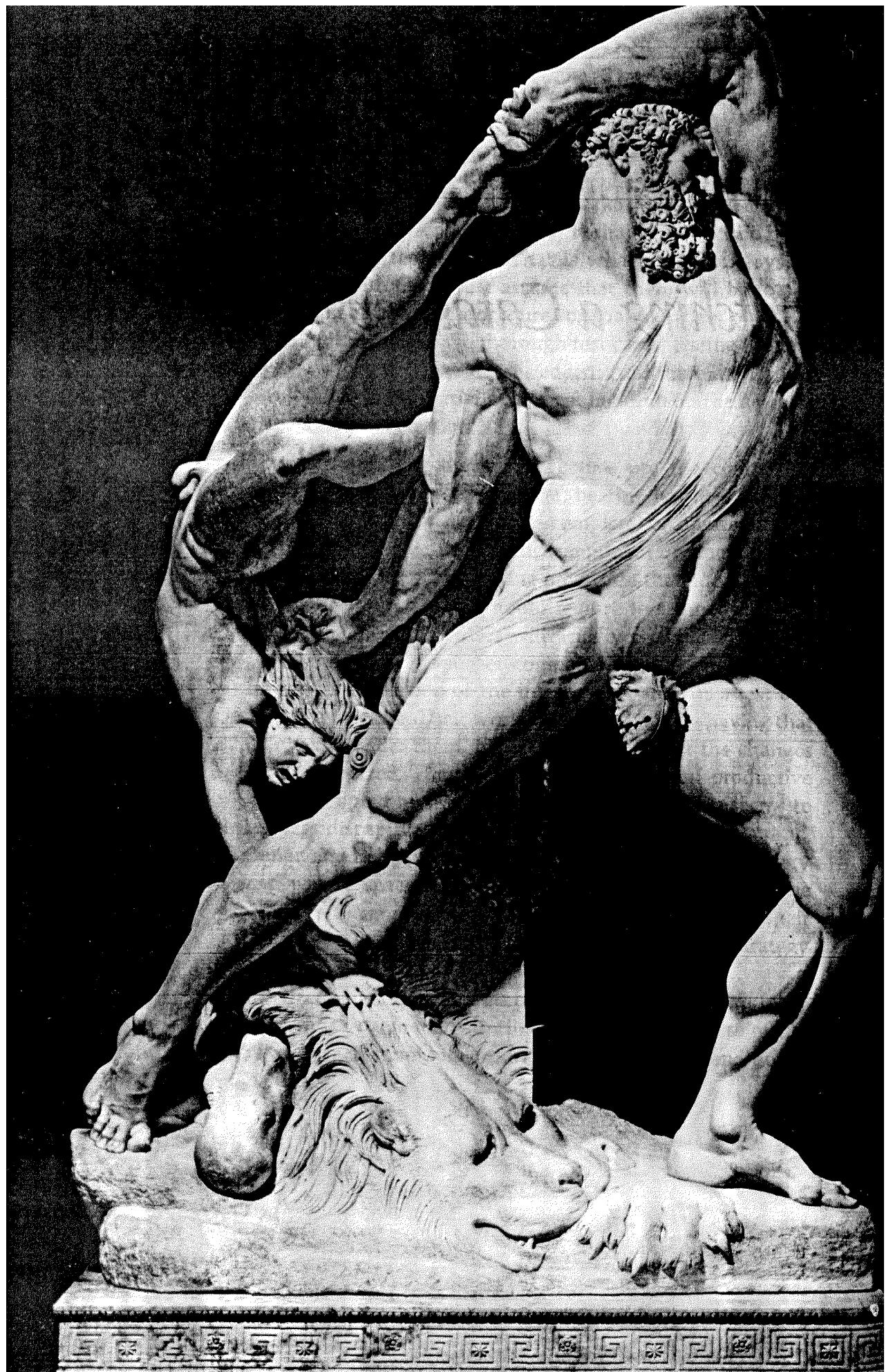
Cada equipe construindo outro componente tem usado a versão testada mais recente do sistema integrado como uma base de teste para depurar sua parte. Seu trabalho será atrasado por ter aquela mudança de banco de ensaio sob eles. Claro que deve. Mas as mudanças precisam ser quantizadas. Então, cada usuário tem períodos de estabilidade produtiva, interrompidos por surtos de mudança de banco de ensaio. Isso parece ser muito menos perturbador do que ondulações e tremores constantes.

Lehman e Belady oferecem evidências de que os quanta devem ser muito grandes e bem espaçados ou então muito pequenos e frequentes.¹⁴ Esta última estratégia está mais sujeita a instabilidade, de acordo com seu modelo. Minha experiência o confirma: eu nunca arriscaria essa estratégia na prática.

As alterações quantizadas acomodam perfeitamente uma técnica de fio roxo. O patch rápido é válido até o próximo lançamento regular do componente, que deve incorporar a correção de forma testada e documentada.

14

Chocando uma catástrofe



14

Chocando uma catástrofe

Ninguém ama o portador de más notícias.

SOFÓCULOS

Como um projeto chega um ano atrasado?

... *Um dia de cada vez.*

A. Canova, "Ercole e lica", 1802. Hércules atira para a morte o mensageiro Lica, que inocentemente trouxe a vestimenta da morte.
Scala / ArtResource, NY

Quando alguém ouve falar de um deslizamento desastroso de cronograma em um projeto, ele imagina que uma série de grandes calamidades deve ter acontecido. Normalmente, no entanto, o desastre é causado por cupins, não tornados; e a programação caiu imperceptivelmente, mas inexoravelmente. Na verdade, grandes calamidades são mais fáceis de lidar; responde-se com grande força, reorganização radical, a invenção de novas abordagens. Toda a equipe está à altura da ocasião.

Mas o deslize diário é mais difícil de reconhecer, mais difícil de prevenir, mais difícil de compensar. Ontem, um homem-chave estava doente e não foi possível realizar uma reunião. Hoje as máquinas estão todas desligadas porque um raio atingiu o transformador de energia do prédio. Amanhã as rotinas de disco não começam o teste, porque o primeiro disco está uma semana atrasado da fábrica. Neve, serviço do júri, problemas familiares, reuniões de emergência com clientes, auditorias executivas - a lista é infinita. Cada um apenas adia alguma atividade por meio dia ou um dia. E a programação falha, um dia de cada vez.

Marcos ou Mós?

Como controlar um grande projeto em um cronograma apertado? O primeiro passo é *tenho* um cronograma. Cada um de uma lista de eventos, chamados de marcos, tem uma data. Escolher as datas é um problema de estimativa, já discutido e crucialmente dependente da experiência.

Para escolher os marcos, existe apenas uma regra relevante. Os marcos devem ser eventos concretos, específicos e mensuráveis, definidos com nitidez de ponta de faca. A codificação, por exemplo, é "90 por cento concluída" durante metade do tempo total de codificação. A depuração está "99% concluída" na maioria das vezes. "Planejamento completo" é um evento que se pode proclamar quase à vontade.¹

Marcos concretos, por outro lado, são eventos 100 por cento. "Especificações assinadas por arquitetos e implementadores", "codificação de origem 100 por cento concluída, keypunched, inserida na biblioteca de disco", "versão depurada passa em todos os casos de teste." Esses marcos concretos marcam as fases vagas de planejamento, codificação, depuração.

É mais importante que os marcos sejam precisos e inequívocos do que facilmente verificáveis pelo chefe. Raramente um homem mentirá sobre o progresso de um marco, *E se* o marco é tão nítido que ele não consegue se enganar. Mas, se a pedra de toque for confusa, o chefe geralmente entende um relato diferente daquele que o homem dá. Para complementar Sófocles, ninguém gosta de levar más notícias, então as notícias são amenizadas sem nenhuma intenção real de enganar.

Dois estudos interessantes de estimativa do comportamento de empreiteiros do governo em projetos de desenvolvimento de grande escala mostram que:

1. As estimativas da duração de uma atividade, feitas e revisadas cuidadosamente a cada duas semanas antes do início da atividade, não mudam significativamente à medida que o horário de início se aproxima, não importa o quão erradas elas acabem sendo.
- 2 *No decorrer* Durante a atividade, as superestimativas da duração diminuem continuamente à medida que a atividade prossegue.
- 3 *Subestima* não mude significativamente durante a atividade até cerca de três semanas antes da conclusão programada.²

Marcos precisos são, na verdade, um serviço para a equipe e que eles podem esperar de um gerente. O marco difuso é o fardo mais difícil de se conviver. Na verdade, é uma pedra de moinho que mói o moral, pois engana sobre o tempo perdido até que se torne irremediável. E o deslize crônico do cronograma é um assassino moral.

"A outra parte está atrasada, de qualquer maneira"

Um cronograma falha por dia; e daí? Quem fica animado com um deslize de um dia? Podemos compensar mais tarde. E a outra peça em que a nossa se encaixa está atrasada, de qualquer maneira.

Um gerente de beisebol reconhece um talento não físico, *labuta*, como um presente essencial para grandes jogadores e grandes equipes. É a característica de correr mais rápido do que o necessário, movendo-se mais cedo do que o necessário, esforçando-se mais do que o necessário. É essencial para grandes equipes de programação também. O Hustle fornece a almofada, a capacidade de reserva, que permite a uma equipe lidar com contratempos rotineiros, para

anticipar e evitar calamidades menores. A resposta calculada, o esforço medido, são os cobertores molhados que amortecem a agitação. Como vimos, um *deve* ficar animado com um deslize de um dia. Esses são os elementos da catástrofe.

Mas nem todos os deslizes de um dia são igualmente desastrosos. Portanto, algum cálculo de resposta é necessário, embora a pressa seja atenuada. Como saber quais deslizes são importantes? Não há substituto para um gráfico PERT ou um cronograma de caminho crítico. Essa rede mostra quem espera o quê. Mostra quem está no caminho crítico, para onde qualquer deslize move a data de término. Também mostra o quanto uma atividade pode escorregar antes de entrar no caminho crítico.

A técnica PERT, a rigor, é uma elaboração de escalonamento de caminho crítico em que se estima três vezes para cada evento, tempos correspondentes a diferentes probabilidades de cumprimento das datas estimadas. Não acho que esse refinamento valha o esforço extra, mas para abreviar, chamarei qualquer rede de caminho crítico de gráfico PERT.

A preparação de um gráfico PERT é a parte mais valiosa de seu uso. Dispor a rede, identificar as dependências e estimar as pernas, tudo isso exige um grande planejamento muito específico no início de um projeto. O primeiro gráfico é sempre terrível, e alguém inventa e inventa ao fazer o segundo.

À medida que o projeto avança, o gráfico PERT fornece a resposta à desculpa desmoralizante: "A outra parte está atrasada de qualquer maneira." Mostra como a pressa é necessária para manter a própria parte fora do caminho crítico e sugere maneiras de compensar o tempo perdido na outra parte.

Sob o tapete

Quando um gerente de primeira linha vê sua pequena equipe ficando para trás, raramente está inclinado a correr para o chefe com esse problema. A equipe pode ser capaz de inventar, ou ele deve ser capaz de inventar ou reorganizar para resolver o problema. Então por que preocupar o chefe com isso? Até agora

Boa. Resolver esses problemas é exatamente o que o gerente de primeira linha existe. E o chefe tem preocupações reais suficientes exigindo sua ação para que ele não busque os outros. Assim, toda a sujeira é varrida para baixo do tapete.

Mas todo chefe precisa de dois tipos de informação, exceções ao plano que requerem ação e um quadro de status para a educação.³ Para isso, ele precisa saber o status de todas as suas equipes. É difícil obter uma imagem real desse status.

Os interesses do gerente de primeira linha e os do chefe têm um conflito inerente aqui. O gerente de primeira linha teme que, se relatar seu problema, o chefe aja. Então, sua ação prejudicará a função do gerente, diminuirá sua autoridade, atrapalhará seus outros planos. Portanto, enquanto o gerente achar que pode resolver o problema sozinho, ele não conta ao chefe.

Duas técnicas de levantamento de tapete estão disponíveis para o chefe. Ambos devem ser usados. O primeiro é reduzir o conflito de papéis e inspirar o compartilhamento de status. A outra é puxar o tapete para trás.

Reduzindo o conflito de funções. O chefe deve primeiro distinguir entre informações de ação e informações de status. Ele deve se disciplinar *não* para agir em problemas que seus gerentes podem resolver, e *nunca* para agir em problemas quando ele está explicitamente revisando o status. Certa vez, conheci um chefe que invariavelmente pegava o telefone para dar ordens antes do final do primeiro parágrafo em um relatório de status. Essa resposta é garantida para impedir a divulgação completa.

Por outro lado, quando o gerente sabe que seu chefe aceitará relatórios de status sem pânico ou preempção, ele passa a fazer avaliações honestas.

Todo este processo é ajudado se o chefe rotular reuniões, revisões, conferências, como *revisão de status* reuniões versus *problema-ação* reuniões e se controla de acordo. Obviamente, alguém pode convocar uma reunião de ação de problema como consequência de uma reunião de status, se acreditar que o problema está fora de controle. Mas pelo menos todo mundo sabe qual é o placar, e o chefe pensa duas vezes antes de pegar a bola.

Arrancando o tapete. No entanto, é necessária a existência de técnicas de revisão através das quais se dê a conhecer o verdadeiro estado, cooperativa ou não. O gráfico PERT com seus marcos freqüentes e nítidos é a base para tal revisão. Em um grande projeto, pode-se querer revisar alguma parte dele a cada semana, fazendo as rondas uma vez por mês mais ou menos.

Um relatório mostrando marcos e conclusões reais é o documento-chave. A Figura 14.1 mostra um trecho desse relatório. Este relatório mostra alguns problemas. A aprovação das especificações está atrasada para vários componentes. A aprovação manual (SLR) está atrasada em outro, e um está atrasado para sair do primeiro estado (Alfa) do teste de produto conduzido de forma independente. Portanto, esse relatório serve de ordem do dia para a reunião de 1 de fevereiro. Todos conhecem as perguntas e o gerente de componentes deve estar preparado para explicar por que é tarde, quando será concluído, quais etapas ele está tomando e que ajuda, se houver, ele precisa do chefe ou dos grupos colaterais.

V. Vyssotsky da Bell Telephone Laboratories acrescenta o seguinte-observação de ing:

*I acharam útil incluir datas "programadas" e "estimadas" no relatório de marcos. As datas programadas são de propriedade do gerente de projeto e representam um plano de trabalho consistente para o projeto como um todo, e que é a priori um plano razoável. As datas estimadas são de propriedade do gerente de nível mais baixo que tem conhecimento sobre o trabalho em questão e representa seu melhor julgamento de quando isso realmente acontecerá, dados os recursos de que dispõe e quando recebeu (ou tem compromissos para entrega de) suas entradas de pré-requisito. O gerente de projeto deve manter seus dedos longe das datas estimadas e colocar ênfase na obtenção de estimativas imparciais e precisas, em vez de estimativas otimistas palatáveis ou conservadoras autoprotetoras. Uma vez que isso esteja claramente estabelecido na mente de todos, o gerente de projeto pode ver muitos caminhos no futuro onde ele terá problemas se não fizer algo. **

PROJECT	LOCATION	COMMITMENT ANNOUNCE RELEASE	OBJECTIVE AVAILABLE APPROVED	SRL AVAILABLE APPROVED	ALPHA ENTRY EXIT	TEST START COMPLETE	SYS TEST	BULLETIN	BETA TEST
							START COMPLETE	ENTRY APPROVED	EXIT APPROVED
OPERATING SYSTEM									
12K DESIGN LEVEL (E)									
ASSEMBLY	SAN JOSE	04/-/-/4 C 12/31/5	10/28/A C 01/11/5	10/13/A C 11/16/A	C A	01/13/A C 02/12/S	01/15/S C 02/22/S	09/01/5 11/30/5	
FORTRAN	POK	04/-/-/4 C 12/31/5	10/28/A C 01/22/5	10/21/A C 12/19/A	C A	12/17/A C 02/22/S	01/15/S C 02/22/S	09/01/5 11/30/5	
COBOL	ENDICOTT 12/31/5	04/-/-/4 C 10/28/A C	10/15/A C 01/20/5 A	11/17/A C 12/06/A	C A	01/17/A C 02/22/S	01/15/S C 02/22/S	09/01/5 11/30/5	
NPL	SAN JOSE	04/-/-/4 C 12/31/5	10/28/A C 01/05/5 A	09/30/A C 01/16/S A	C A	12/02/A C 02/22/S	01/15/S C 02/22/S	09/01/5 11/30/5	
UTILITIES	TIME/LIFE	04/-/-/4 C 12/31/5	06/24/A C 11/30/4	11/20/A C 11/30/A	C A	11/20/A C 11/30/A	01/15/S C 02/22/S	09/01/5 11/30/5	
SORT 1	POK	04/-/-/4 C 12/31/5	10/28/A C 01/11/5	10/19/A C 11/30/A	C A	11/12/A C 03/22/S	01/15/S C 03/22/S	09/01/5 11/30/5	
SORT 2	POK	04/-/-/4 C 06/30/6	10/28/A C 01/11/5	10/19/A C 11/30/A	C A	11/12/A C 03/22/S	01/15/S C 03/22/S	09/01/6 05/30/6	
44K DESIGN LEVEL (F)									
ASSEMBLY	SAN JOSE	04/-/-/4 C 12/31/5	10/28/A C 01/11/5	10/13/A C 11/16/A	C A	11/13/A C 03/22/S	02/15/S C 03/22/S	09/01/5 11/30/5	
COBOL	TIME/LIFE	04/-/-/4 C 06/30/6	10/28/A C 01/20/5	10/15/A C 12/08/A	C A	11/12/A C 03/22/S	02/15/S C 03/22/S	09/01/5 05/30/6	
NPL	HURSLEY	04/-/-/4 C 03/31/6	10/28/A C 11/30/5	11/18/A C 01/04/5	C A	11/17/A C 01/29/5	01/04/5 C 04/30/5	01/03/6 01/28/6	
2250	KINGSTON	03/30/A C 03/31/6	11/05/A C 01/04/5	12/03/A C 01/04/5	C A	01/12/5 C 01/29/5	01/04/5 C 04/30/5	01/03/6 01/28/6	
2280	KINGSTON	06/30/A C 09/30/6	11/05/A C 01/04/5	11/12/A C 01/04/5	C A	11/12/A C 01/04/5	01/04/5 C 04/30/5	01/03/6 01/28/6	
200K DESIGN LEVEL (H)									
ASSEMBLY	TIME/LIFE								
FORTRAN	POK	04/-/-/4 C 06/30/6	10/28/A C 01/11/5	10/16/A C 12/10/A	C A	11/16/A C 03/22/S	02/15/S C 03/22/S	03/01/6 05/30/6	
NPL	HURSLEY	04/-/-/4 C 03/31/7	10/28/A C 03/31/7	10/28/A C 03/31/7	C A	01/--/5 C 01/29/5	01/--/5 C 01/29/5	01/--/7 01/29/5	
NPL H	POK	04/-/-/4 C 03/30/A	03/30/A C 04/01/5	03/30/A C 04/01/5	C A	02/01/5 C 04/01/5	02/01/5 C 04/01/5	10/15/S 12/15/S	

Figure 14.1

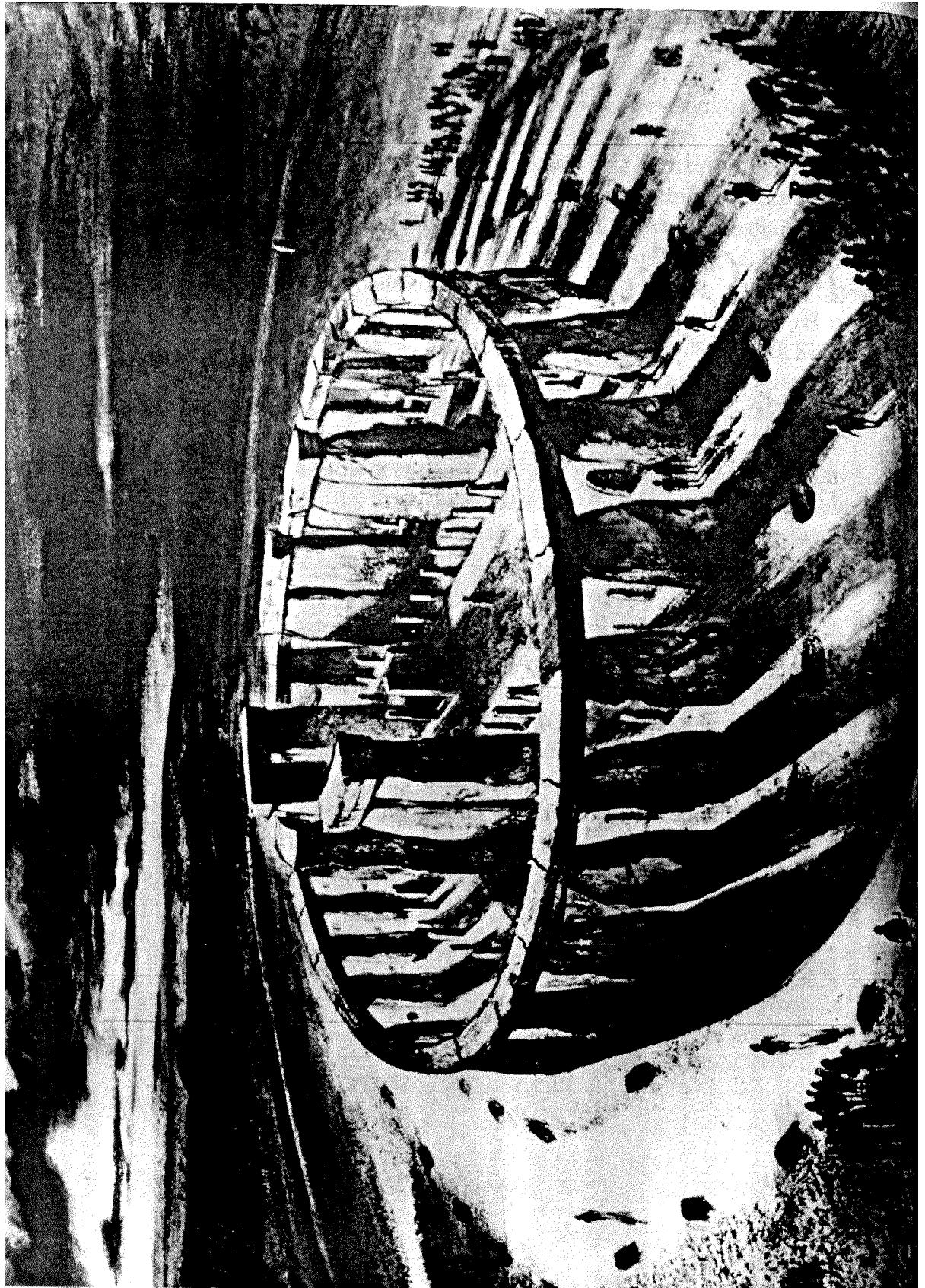
A preparação do gráfico PERT é função do chefe e dos gerentes que se reportam a ele. Sua atualização, revisão e relatório requerem a atenção de um pequeno grupo de funcionários (de um a três homens) que serve como uma extensão do chefe. Tal *Planos e controles* equipe é inestimável para um grande projeto. Não tem autoridade, exceto para perguntar a todos os gerentes de linha quando eles terão estabelecido ou alterado os marcos e se os marcos foram cumpridos. Como o grupo de Planos e Controles cuida de toda a papelada, a carga dos gerentes de linha é reduzida ao essencial - tomar as decisões.

Tínhamos um grupo de Planos e Controles habilidoso, entusiasta e diplomático, dirigido por AM Pietrasanta, que dedicou considerável talento inventivo para desenvolver métodos de controle eficazes, mas discretos. Como resultado, descobri que seu grupo era amplamente respeitado e mais do que tolerado. Para um grupo cujo papel é inherentemente o de um irritante, isso é uma grande conquista.

O investimento de uma quantidade modesta de esforço qualificado em uma função de Planos e Controles é muito gratificante. Isso faz com que a diferença na realização do projeto seja maior do que se essas pessoas trabalhassem diretamente na construção dos programas do produto. Para o grupo Planos e Controles é o cão de guarda que torna visíveis os atrasos imperceptíveis e que aponta os elementos críticos. É o sistema de alerta precoce contra a perda de um ano, um dia de cada vez.

quinze

A outra face



quinze

A outra face

O que não entendemos, não possuímos.

GOETHE

*Ó, dê-me comentaristas simples,
Quem sem pesquisas profundas irrita o cérebro.*

CRABBE

Uma reconstrução de Stonehenge, o maior computador não documentado do mundo.

Arquivo Bettman

Um programa de computador é uma mensagem de um homem para uma máquina. A sintaxe rigidamente organizada e as definições escrupulosas existem para tornar a intenção clara para o motor estúpido.

Mas um programa escrito tem outra cara, aquela que conta sua história para o usuário humano. Mesmo para o mais privado dos programas, alguma comunicação desse tipo é necessária; a memória falhará ao autor-usuário, e ele precisará se atualizar sobre os detalhes de sua obra.

Quão mais vital é a documentação de um programa público, cujo usuário está distante do autor tanto no tempo quanto no espaço! Para o produto do programa, a outra face para o usuário é tão importante quanto a face para a máquina.

A maioria de nós criticou discretamente o autor remoto e anônimo de algum programa pouco documentado. E muitos de nós tentamos, portanto, incutir nos novos programadores uma atitude sobre a documentação que os inspiraria para o resto da vida, superando a preguiça e a pressão de cronograma. Em geral, falhamos. Acho que usamos métodos errados.

Thomas J. Watson, Sr. contou a história de sua primeira experiência como vendedor de caixa registradora no interior do estado de Nova York. Carregado de entusiasmo, ele saiu com sua carroça carregada de caixas registradoras. Ele trabalhou seu território diligentemente, mas sem vender um. Abatido, relatei a seu chefe. O gerente de vendas ouviu um pouco e disse: "Ajude-me a carregar alguns registros na carroça, atrelar o cavalo e vamos de novo". Eles o fizeram, e os dois chamaram cliente após cliente, com o homem mais velho *mostrando como* para vender caixas registradoras. Todas as evidências indicam que a lição foi concluída.

Por vários anos, lecionei diligentemente em minha aula de engenharia de software sobre a necessidade e a propriedade de uma boa documentação, exortando-os cada vez com mais fervor e eloquência. Não funcionou. Presumi que eles haviam aprendido a documentar adequadamente e estavam falhando por falta de zelo. Então tentei carregar algumas caixas registradoras no vagão; ou seja, *mostrando* eles como o trabalho é feito. Isso teve muito mais sucesso. Portanto, o restante deste ensaio minimizará a exortação e se concentrará no "como" de uma boa documentação.

Qual documentação é necessária?

Diferentes níveis de documentação são necessários para o usuário ocasional de um programa, para o usuário que deve depender de um programa e para o usuário que deve adaptar um programa para mudanças nas circunstâncias ou no propósito.

Para usar um programa. Cada usuário precisa de uma descrição em prosa do programa. A maioria da documentação falha em fornecer muito pouca visão geral. As árvores são descritas, a casca e as folhas são comentadas, mas não há mapa da floresta. Para escrever uma descrição útil em prosa, afaste-se e entre devagar:

- 1 *Propósito.* Qual é a função principal, o motivo do programa?
- 2 *Ambiente.* Em quais máquinas, configurações de hardware e configurações de sistema operacional ele será executado?
- 3 *Domínio e alcance.* Que domínio de entrada é válido? Que faixa de saída pode aparecer legitimamente?
- 4 *Funções realizadas e algoritmos usados.* O que exatamente isso faz?
- 5 *Formatos de entrada-saída*, preciso e completo.
- 6 *Instruções de operação*, incluindo comportamento final normal e anormal, como visto no console e nas saídas.
- 7 *Opções* Que opções o usuário tem sobre as funções? Exatamente como essas opções são especificadas?
- 8 *Tempo de execução.* Quanto tempo leva para resolver um problema de tamanho especificado em uma configuração especificada?
- 9 *Precisão e verificação.* Quão precisas são as respostas esperadas? Quais meios de verificação de precisão são incorporados?

Freqüentemente, todas essas informações podem ser apresentadas em três ou quatro páginas. Isso requer muita atenção à concisão e precisão. A maior parte deste documento precisa ser redigida antes de o programa ser escrito, pois ele incorpora decisões básicas de planejamento.

Para acreditar em um programa. A descrição de como ele é usado deve ser complementada com alguma descrição de como se sabe que está funcionando. Isso significa casos de teste.

Cada cópia de um programa enviado deve incluir alguns pequenos casos de teste que podem ser usados rotineiramente para assegurar ao usuário que ele tem uma cópia fiel, carregada com precisão na máquina.

Em seguida, são necessários casos de teste mais completos, que normalmente são executados somente depois que um programa é modificado. Eles se enquadram em três partes do domínio de dados de entrada:

1. Mainlinecases que testam as chieffunctions do programa para dados comumente encontrados.
2. Casos dificilmente legítimos que investigam a borda do domínio de dados de entrada, garantindo que os maiores valores possíveis, os menores valores possíveis e todos os tipos de exceções válidas funcionem.
3. Casos quase ilegítimos que investigam os limites do domínio do outro lado, garantindo que entradas inválidas gerem mensagens de diagnóstico adequadas.

Para modificar um programa. Adaptar um programa ou consertá-lo requer muito mais informações. É claro que todos os detalhes são necessários, e isso está contido em uma lista bem comentada. Para o modificador, assim como para o usuário mais casual, a necessidade premente é de uma visão geral clara e nítida, desta vez da estrutura interna. Quais são os componentes dessa visão geral?

1. Um fluxograma ou gráfico de estrutura de subprograma. Mais sobre isso mais tarde.
2. Descrições completas dos algoritmos usados, ou então referências a tais descrições na literatura.
3. Uma explicação do layout de todos os arquivos usados.
4. Uma visão geral da estrutura da passagem - a sequência na qual os dados ou programas são trazidos da fita ou disco - e o que é realizado em cada passagem.
5. Uma discussão das modificações contempladas no projeto original, a natureza e localização dos ganchos e saídas, e uma discussão discursiva das idéias do autor original sobre quais modificações podem ser desejáveis e como elas podem proceder. Suas observações sobre armadilhas ocultas também são úteis.

A maldição do fluxograma

O fluxograma é uma peça completamente exagerada da documentação do programa. Muitos programas não precisam de fluxogramas; poucos programas precisam de mais do que um fluxograma de uma página.

Os fluxogramas mostram a estrutura de decisão de um programa, que é apenas um aspecto de sua estrutura. Eles mostram a estrutura de decisão de forma bastante elegante quando o fluxograma está em uma página, mas o

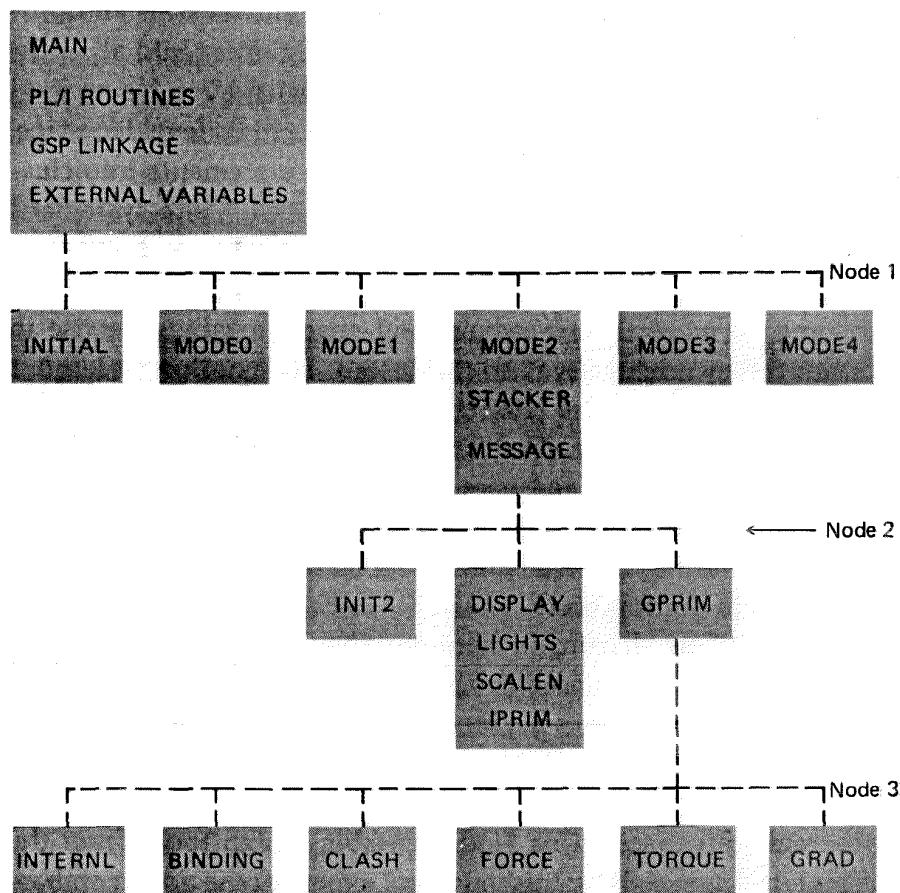


Fig. 15.1 Um gráfico da estrutura do programa. (Cortesia de WV Wright)

a visualização se quebra muito quando se tem várias páginas, costuradas com saídas e conectores numerados.

O fluxograma de uma página para um programa substancial torna-se essencialmente um diagrama da estrutura do programa e de fases ou etapas. Como tal, é muito útil. A Figura 15.1 mostra esse gráfico de estrutura de subprograma.

É claro que esse gráfico de estrutura não segue nem precisa dos padrões de fluxograma ANSI penosamente elaborados. Todas as regras sobre formatos de caixa, conectores, numeração, etc. são necessários apenas para dar inteligibilidade a fluxogramas detalhados.

O fluxograma detalhado passo a passo, no entanto, é um incômodo obsoleto, adequado apenas para iniciantes no pensamento algorítmico. Quando apresentado por Goldstine e von Neumann,¹

as caixinhas e seu conteúdo serviam como uma linguagem de alto nível, agrupando as inescrutáveis declarações da linguagem de máquina em grupos de significados. Como Iverson reconheceu cedo,² em uma linguagem sistemática de alto nível, o agrupamento já está feito e cada caixa contém uma declaração (Fig. 15.2). Então, as próprias caixas se tornam nada mais do que um exercício tedioso e monótono de desenho; eles também podem ser eliminados. Então nada resta além das flechas. As setas que unem uma instrução a seu sucessor são redundantes; apague-os. Isso deixa apenas GO TO's. E se seguirmos as boas práticas e usarmos a estrutura de blocos para minimizar os GO TO, não haverá muitas setas, mas ajudam imensamente a compreensão. Uma noite também os desenha na lista e elimina o fluxograma por completo.

Na verdade, o fluxograma é mais pregado do que praticado. Nunca vi um programador experiente que fizesse rotineiramente fluxogramas detalhados antes de começar a escrever programas. Onde os padrões da organização exigem fluxogramas, estes são quase sempre feitos após o fato. Muitas lojas usam orgulhosamente programas de máquina para gerar essa "ferramenta de projeto indispensável" a partir do código concluído. Acho que essa experiência universal não é um afastamento constrangedor e deplorável das boas práticas, que só pode ser reconhecida com uma risada nervosa. Em vez disso, é o

aplicação do bom senso e nos ensina algo sobre a utilidade dos fluxogramas.

O apóstolo Pedro disse a respeito dos novos convertidos gentios e da lei judaica: "Por que colocar nas costas [deles] uma carga que nem nossos ancestrais nem nós mesmos fomos capazes de carregar?" (Atos 15:10, TEV). Eu diria o mesmo sobre novos programadores e a prática obsoleta de fluxogramas.

Programas autodocumentáveis

Um princípio básico de processamento de dados ensina a loucura de tentar manter arquivos independentes em sincronismo. É muito melhor combiná-los em um arquivo com cada registro contendo todas as informações que ambos os arquivos possuem sobre uma determinada chave.

No entanto, nossa prática na documentação de programação viola nosso próprio ensino. Normalmente tentamos manter uma forma de programa legível por máquina e um conjunto independente de documentação legível por humanos, consistindo em prosa e fluxogramas.

Os resultados de fato confirmam nossos ensinamentos sobre a loucura de arquivos separados. A documentação do programa é notoriamente pobre e sua manutenção é pior. As alterações feitas no programa não aparecem pronta, precisa e invariavelmente no papel.

A solução, eu acho, é mesclar os arquivos, para incorporar a documentação no programa de origem. Isso é ao mesmo tempo um poderoso incentivo para a manutenção adequada e uma garantia de que a documentação estará sempre à mão para o usuário do programa. Esses programas são chamados *autodocumentado*.

Agora, claramente, isso é estranho (mas não impossível) se os fluxogramas forem incluídos. Mas garantindo a obsolescência dos fluxogramas e o uso dominante de linguagem de alto nível, torna-se razoável combinar o programa e a documentação.

O uso de um programa fonte como meio de documentação impõe algumas restrições. Por outro lado, a disponibilidade íntima do programa fonte, linha por linha, ao leitor da documentação possibilita novas técnicas. O tempo tem

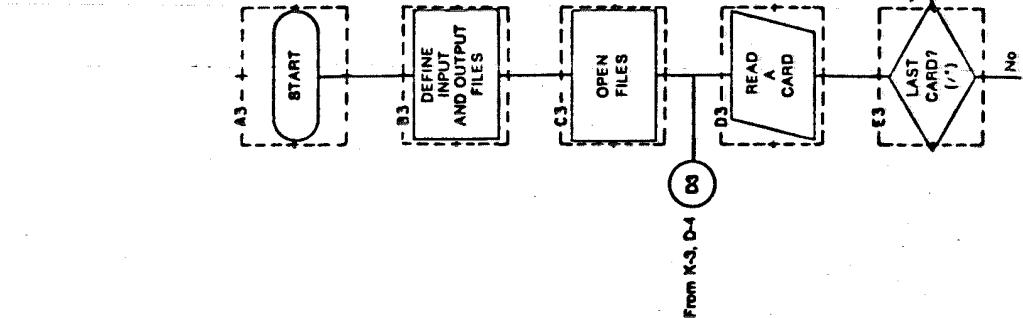
PCRS: PROCEDURE OPTIONS (MAIN):

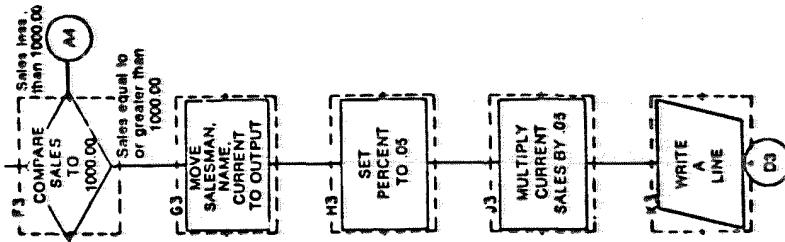


```

DECLARE SALEFL FILE
  RECORD
    INPUT ENVIRONMENT (F180) MEDIUM (SYSIPT, 2501);
  RECORD
    OUTPUT ENVIRONMENT (F1132) MEDIUM (SYSLST, 1403) CTLASAT;
  RECORD
    SALESARD,
      03 BLANK1           CHARACTER 191,
      03 SALENUM           PICTURE '99999',
      03 NAME              CHARACTER 1251,
      03 BLANK2           CHARACTER 171,
      03 CURRENT_SALES     PICTURE '9999999',
      03 BLANK3           CHARACTER 1291;
  RECORD
    SALESLIST,
      03 CONTROL           CHARACTER 111 INITIAL (' '),
      03 SALENUM_OUT        PICTURE '22299',
      03 FILLER1            CHARACTER 151 INITIAL (' '),
      03 NAME_OUT           CHARACTER 1251,
      03 FILLER2            CHARACTER 151 INITIAL (' '),
      03 CURRENT_OUT         PICTURE '22222V99',
      03 FILLER3            CHARACTER 151 INITIAL (' '),
      03 PERCENT            PICTURE '29',
      03 SIGN               CHARACTER 111 INITIAL (''''),
      03 FILLER4            CHARACTER 151 INITIAL (' '),
      03 COMMISSION          PICTURE '22222V99',
      03 FILLERS             CHARACTER 1631 INITIAL (' '));
OPEN FILE (SALEFL).FILE (PRINT4);
ON ENDFILE (SALEFL) GO TO ENDJOB;

```





```

READ_CARD:
    READ FILE ISALEFL) INTO ISALESCARD;
    IF CURRENT_SALES < 1000.00 THEN GO TO UNDER_QUOTA;
    SALESNUM=OUT-SALESNUM;
    NAME_OUT=NAME;
    CURRENT_OUT=CURRENT_SALES;
    PERCENT=.05;
    COMMISSION=CURRENT_SALES*.05;
    WRITE FILE (PRIN14) FROM ISALESLIST1;
    GO TO READ_CARD;

```

Fig. 15.2 Comparison of a flow chart and a corresponding PL/I program.
 [Abridged and adapted from Figs. 15-41, 15-44, in *Data Processing and Computer Programming: A Modular Approach* by Thomas J. Cashman and William J. Keys (Harper & Row, 1971).]

chegam a conceber abordagens e métodos radicalmente novos para a documentação do programa.

Como objetivo principal, devemos tentar minimizar o fardo da documentação, o fardo que nem nós nem nossos predecessores fomos capazes de suportar com sucesso.

Uma abordagem. A primeira noção é usar as partes do programa que devem estar lá de qualquer maneira, por razões de linguagem de programação, para carregar o máximo de documentação possível. Assim, rótulos, declarações de declarações e nomes simbólicos são todos atrelados à tarefa de transmitir o máximo de significado possível ao leitor.

Uma segunda noção é usar o espaço e o formato o máximo possível para melhorar a legibilidade e mostrar a subordinação e o aninhamento.

A terceira noção é inserir a documentação necessária em prosa no programa como parágrafos de comentários. A maioria dos programas tende a ter comentários linha por linha suficientes; Esses programas produzidos para atender a rígidos padrões organizacionais de "boa documentação" geralmente têm muitos. Todos esses programas, entretanto, são geralmente deficientes nos comentários de parágrafo que realmente dão inteligibilidade e visão geral para a coisa toda.

Uma vez que a documentação é incorporada à estrutura, nomenclatura e formatos do programa, grande parte dela *deve* ser feito quando o programa é escrito pela primeira vez. Mas isso é quando *deve* ser escrito. Como a abordagem de autodocumentação minimiza o trabalho extra, há menos obstáculos para fazê-lo.

Algumas técnicas. A Figura 15.3 mostra um programa PL / I autodocumentado.³ Os números nos círculos não fazem parte dela; eles são meta-documentação ligada à discussão.

1. Use um nome de trabalho separado para cada execução e mantenha um registro de execução mostrando o que foi tentado, quando e os resultados. Se o nome for composto por uma parte mnemônica (aqui *QLT*) e um sufixo numérico (aqui *4*), o sufixo pode ser usado como um número de execução, vinculando listagens e log. Essa técnica requer um novo cartão de trabalho para cada execução, mas eles podem ser feitos em lotes, duplicando as informações comuns.

Fig. 15.3 Um programa de autodocumentação.



① //QLT4 JOB ...

② QLTSRT7: PROCEDURE (V);

```
*****  
③ /*A SORT SUBROUTINE FOR 2500 6-BYTE FIELDS, PASSED AS THE VECTOR V. A */  
/*SEPARATELY COMPILED, NOT-MAIN PROCEDURE, WHICH MUST USE AUTOMATIC CORE */  
/*ALLOCATION.  
/*  
④ /*THE SORT ALGORITHM FOLLOWS BROOKS AND IVERSON, AUTOMATIC DATA PROCESSING. */  
/*PROGRAM 7.23, P. 350. THAT ALGORITHM IS REVISED AS FOLLOWS:  
⑤ /* STEPS 2-12 ARE SIMPLIFIED FOR M=2.  
/* STEP 18 IS EXPANDED TO HANDLE EXPLICIT INDEXING OF THE OUTPUT VECTOR.  
/* THE WHOLE FIELD IS USED AS THE SORT KEY.  
/* MINUS INFINITY IS REPRESENTED BY ZEROS.  
/* PLUS INFINITY IS REPRESENTED BY ONES.  
/* THE STATEMENT NUMBERS IN PROG. 7.23 ARE REFLECTED IN THE STATEMENT  
/* LABELS OF THIS PROGRAM.  
/* AN IF-THEN-ELSE CONSTRUCTION REQUIRES REPETITION OF A FEW LINES.  
/*  
/*TO CHANGE THE DIMENSION OF THE VECTOR TO BE SORTED, ALWAYS CHANGE THE  
/*INITIALIZATION OF T. IF THE SIZE EXCEEDS 4096, CHANGE THE SIZE OF T, TOO.  
/*A MORE GENERAL VERSION WOULD PARAMETERIZE THE DIMENSION OF V.  
/*  
/*THE PASSED INPUT VECTOR IS REPLACED BY THE REORDERED OUTPUT VECTOR.  
*****
```

⑥ /* LEGEND (ZERO-ORIGIN INDEXING) */

```
DECLARE  
    (I,           /*INDEX FOR INITIALIZING T */  
     J,           /*INDEX OF ITEM TO BE REPLACED */  
     K,           /*INITIAL INDEX OF BRANCHES FROM NODE I */  
     K) BINARY FIXED, /*INDEX IN OUTPUT VECTOR */  
  
    (MINF,        /*MINUS INFINITY */  
     PINF) BIT (48), /*PLUS INFINITY */  
  
    V (*) BIT (*), /*PASSED VECTOR TO BE SORTED AND RETURNED */  
  
    T (0:8190) BIT (48); /*WORKSPACE CONSISTING OF VECTOR TO BE SORTED, FILLED*/  
    /*OUT WITH INFINITIES, PRECEDED BY LOWER LEVELS */  
    /*FILLED UP WITH MINUS INFINITIES */
```

```
/* NOW INITIALIZATION TO FILL DUMMY LEVELS, TOP LEVEL, AND UNUSED PART OF TOP*/  
/* LEVEL AS REQUIRED. */
```

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

```
DO L= 0 TO 4094; T(L)= MINF; END;  
DO L= 0 TO 2899; T(L+4095)= V(L); END;  
DO L=6595 TO 8190; T(L)= PINF; END;
```

⑧ K0: K = -1;
K1: I = 0; /*
K3: J = 2*I+1; /*SET J TO SCAN BRANCHES FROM NODE I. ⑪
K7: IF T(J) <= T(J+1) /*PICK SMALLER BRANCH
 THEN /*
 ⑨ DO: /*
 K11: T(I) = T(J); /*REPLACE
 K13: IF T(I) = PINF THEN GO TO K16; /*IF INFINITY, REPLACEMENT IS FINISHED
 K12: I = J; /*SET INDEX FOR HIGHER LEVEL
 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; /*GO BACK IF NOT ON TOP LEVEL
 K15: T(I) = PINF; /*IF TOP LEVEL, FILL WITH INFINITY
 K16: IF T(0) = PINF THEN RETURN; /*TEST END OF SORT
 K17: IF T(0) = MINF THEN GO TO K1; /*FLUSH OUT INITIAL DUMMIES
 K18: K = K+1; /*STEP STORAGE INDEX
 V(K) = T(0); GO TO K1; ⑫ /*STORE OUTPUT ITEM
END QLTSRT7;

⑩

2. Use um nome de programa que seja mnemônico, mas também contenha um identificador de versão. Ou seja, suponha que haverá várias versões. Aqui, o índice é o dígito de ordem inferior do ano de 1967.
3. Incorpore a descrição da prosa como comentários ao PROCEDIMENTO.
4. Consulte a literatura padrão para documentar algoritmos básicos sempre que possível. Isso economiza espaço, geralmente aponta para um tratamento muito mais completo do que aquele que forneceria e permite que o leitor experiente o ignore com a certeza de que o comprehende.
5. Mostre a relação com o algoritmo do livro:
 - a) mudanças
 - b) especialização
 - c) representação
6. Declare todas as variáveis. Nomes usuais. Use comentários para converter DECLARE em uma legenda completa. Observe que ele já contém nomes e descrições estruturais, só precisa ser aumentado com descrições de *propósito*. Fazendo isso aqui, pode-se evitar repetir os nomes e as descrições estruturais em um tratamento separado.
7. Marque a inicialização com uma etiqueta.
8. Rotule as declarações em grupos para mostrar que correspondem às declarações na descrição do algoritmo na literatura.
9. Use recuo para mostrar a estrutura e o agrupamento.
10. Adicione setas de fluxo lógico à lista manualmente. Eles são muito úteis na depuração e alteração. Eles podem ser incorporados à margem direita do espaço de comentários e fazer parte do texto legível por máquina.
11. Use comentários de linha ou comente qualquer coisa que não seja óbvia. Se as técnicas acima foram usadas, elas serão curtas e em menor número do que o normal.
12. Coloque várias declarações em uma linha, ou uma declaração em várias linhas para coincidir com o agrupamento de pensamentos e para mostrar a correspondência com a descrição de outro algoritmo.

Por que não? Quais são as desvantagens de tal abordagem para documentação? Existem vários, que foram reais, mas estão se tornando imaginários com a mudança dos tempos.

A objeção mais séria é o aumento no tamanho do código-fonte que deve ser armazenado. À medida que a disciplina se move cada vez mais em direção ao armazenamento on-line de código-fonte, isso se torna uma consideração crescente. Eu me vejo sendo mais breve nos comentários a um programa APL, que viverá em um disco, do que em um programa PL / I que irei armazenar como cartões.

No entanto, simultaneamente, estamos avançando também para o armazenamento on-line de documentos em prosa para acesso e atualização por meio da edição de texto computadorizada. Como mostrado acima, unindo prosa e programa *você reduz* o número total de caracteres a serem armazenados.

Uma resposta semelhante se aplica ao argumento de que programas autodocumentados requerem mais pressionamentos de tecla. Um documento digitado requer pelo menos uma tecla por caractere por rascunho. Um programa de autodocumentação tem menos caracteres no total e também menos traços por caractere, uma vez que os rascunhos não são redigitados.

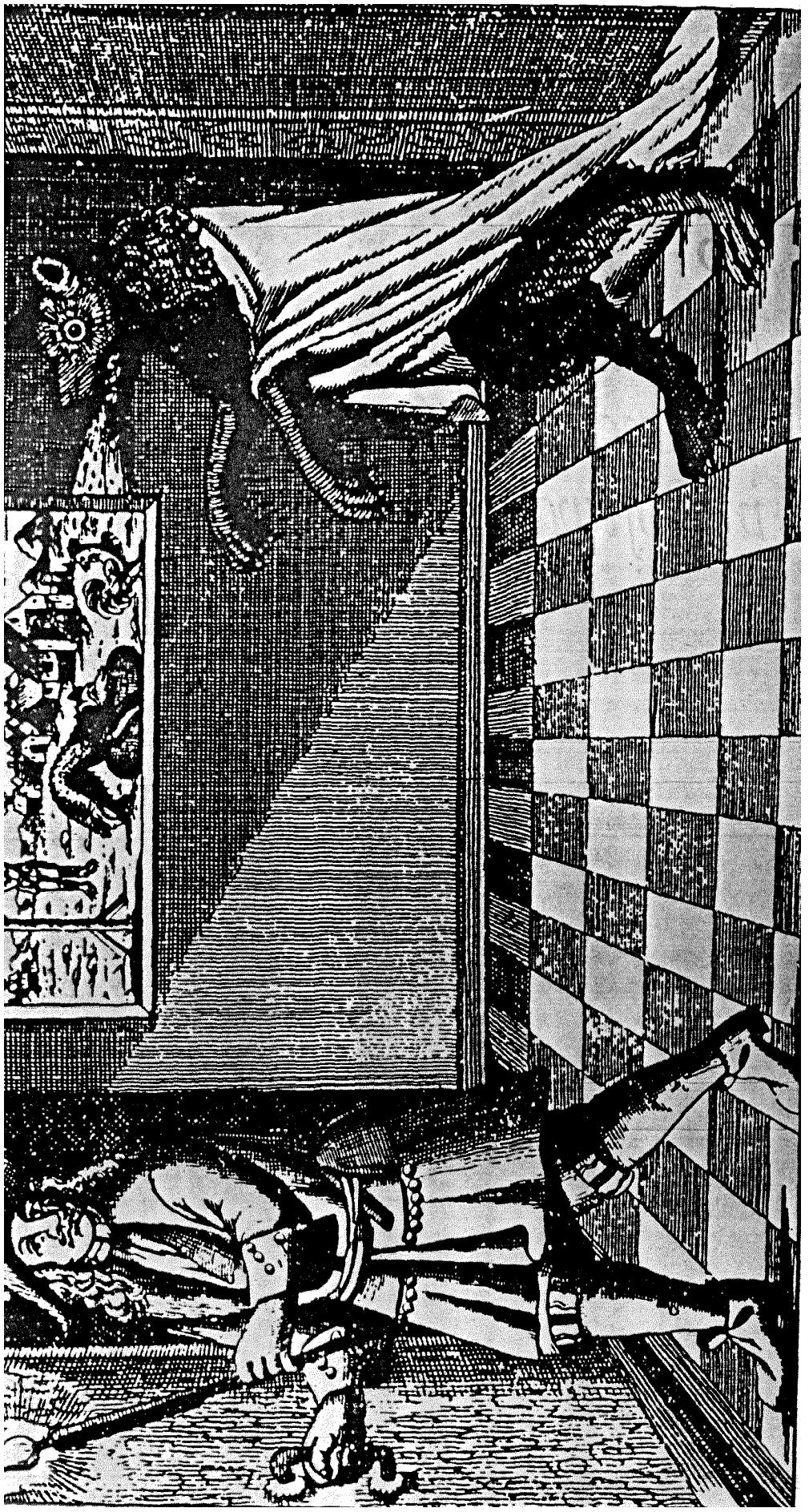
Que tal fluxogramas e gráficos de estrutura? Se alguém usar apenas um gráfico de estrutura de nível mais alto, ele pode ser mantido com segurança como um documento separado, pois não está sujeito a alterações frequentes. Mas certamente pode ser incorporado ao programa de origem como um comentário, e isso parece sensato.

Até que ponto as técnicas usadas acima são aplicáveis a programas em linguagem assembly? Acho que a abordagem básica da autodocumentação é totalmente aplicável. O espaço e os formatos são menos livres e, portanto, não podem ser usados com tanta flexibilidade. Certamente, nomes e declarações estruturais podem ser explorados. As macros podem ajudar muito. O uso extensivo de comentários de parágrafo é uma boa prática em qualquer idioma.

Mas a abordagem de autodocumentação é estimulada pelo uso de linguagens de alto nível e encontra seu maior poder e sua maior justificativa em linguagens de alto nível usadas em sistemas on-line, sejam em lote ou interativos. Como argumentei, essas linguagens e sistemas ajudam os programadores de maneiras muito poderosas. Como as máquinas são feitas para pessoas, não pessoas para máquinas, seu uso faz todo o sentido, econômico e humano.

16

*Sem bala de prata-
Essência e Acidente
em Engenharia de Software*



16

Sem bala de prata- Essência e Acidente em Engenharia de Software

Não existe um desenvolvimento único, seja em tecnologia ou técnica de gestão, o que por si só promete uma melhoria da ordem de grandeza em uma década em produtividade, confiabilidade e simplicidade.

O lobisomem de Eschenbach, Alemanha: gravura em linha, 1685. Cortesia da coleção Grainger, Nova York.

Resumo 1

Toda construção de software envolve tarefas essenciais, a modelagem das estruturas conceituais complexas que compõem a entidade abstrata de software, e tarefas acidentais, a representação dessas entidades abstratas em linguagens de programação e o mapeamento delas em linguagens de máquina dentro de espaço e velocidade restrições. A maioria dos grandes ganhos anteriores em produtividade de software veio da remoção de barreiras artificiais que tornaram as tarefas acidentais excessivamente difíceis, como severas restrições de hardware, linguagens de programação inadequadas, falta de tempo de máquina. Quanto do que os engenheiros de software fazem agora ainda se dedica ao acidental, em oposição ao essencial? A menos que seja mais do que 9/10 de todo o esforço, reduzir todas as atividades acidentais ao tempo zero não proporcionará uma melhoria de ordem de magnitude.

Portanto, parece que chegou a hora de abordar as partes essenciais da tarefa de software, aquelas relacionadas com a criação de estruturas conceituais abstratas de grande complexidade. Eu sugiro:

- Explorar o mercado de massa para evitar construir o que pode ser comprado.
- Usar prototipagem rápida como parte de uma iteração planejada no estabelecimento de requisitos de software.
- Desenvolver software organicamente, adicionando mais e mais funções aos sistemas à medida que são executados, usados e testados.
- Identificar e desenvolver os grandes designers conceituais da nova geração.

Introdução

De todos os monstros que preenchem os pesadelos de nosso folclore, nenhum aterroriza mais do que os lobisomens, porque eles se transformam inesperadamente de familiares em horrores. Para estes, buscamos balas de prata que podem magicamente colocá-los para descansar.

O projeto de software familiar tem algo desse tipo (pelo menos como visto pelo gerente não técnico), geralmente inocente

e direto, mas capaz de se tornar um monstro de cronogramas perdidos, orçamentos estourados e produtos com falhas. Assim, ouvimos gritos desesperados por uma bala de prata, algo que faça os custos de software caírem tão rapidamente quanto os custos de hardware de computador.

Mas, quando olhamos para o horizonte daqui a uma década, não vemos nenhuma bala de prata. Não existe um desenvolvimento único, seja em tecnologia ou técnica de gestão, o que por si só promete uma melhoria de uma ordem de magnitude na produtividade, na confiabilidade, na simplicidade. Neste capítulo, tentaremos ver por quê, examinando tanto a natureza do problema de software quanto as propriedades dos marcadores propostos.

Ceticismo não é pessimismo, entretanto. Embora não vejamos avanços surpreendentes e, na verdade, acreditemos que sejam inconsistentes com a natureza do software, muitas inovações encorajadoras estão em andamento. Um esforço disciplinado e consistente para desenvolvê-los, propagá-los e explorá-los deve, de fato, gerar uma melhoria na ordem de grandeza. Não existe uma estrada real, mas existe uma estrada.

O primeiro passo para o manejo da doença foi a substituição das teorias demoníacas e do humor pela teoria dos germes. Aquele mesmo passo, o início da esperança, em si mesmo frustrou todas as esperanças de soluções mágicas. Dizia aos obreiros que o progresso seria feito gradativamente, com grande esforço, e que um cuidado persistente e incessante deveria ser dispensado à disciplina de limpeza. Assim é com a engenharia de software hoje.

Tem que ser difícil? - Dificuldades essenciais

Não apenas não há balas de prata agora em vista, como a própria natureza do software torna improvável que haja alguma - nenhuma invenção que faça pela produtividade, confiabilidade e simplicidade do software o que a eletrônica, os transistores e a integração em grande escala fizeram pela hardware de computador. Não podemos esperar ver ganhos duplos a cada dois anos.

Em primeiro lugar, devemos observar que a anomalia não é que o progresso do software seja tão lento, mas que o progresso do hardware do computador seja tão

velozes. Nenhuma outra tecnologia desde o início da civilização viu seis ordens de magnitude de ganho de preço-desempenho em 30 anos. Em nenhuma outra tecnologia pode-se escolher obter o ganho em *qualquer* performance melhorada *ou* em custos reduzidos. Esses ganhos fluem da transformação da manufatura de computadores de uma indústria de montagem em uma indústria de processo.

Em segundo lugar, para ver que taxa de progresso podemos esperar na tecnologia de software, vamos examinar suas dificuldades. Seguindo Aristóteles, eu os divido em *essência* —As dificuldades inerentes à natureza do software - e *acidentes* —As dificuldades que hoje acompanham a sua produção, mas que não são inerentes.

Os acidentes são discutidos na próxima seção. Primeiro, vamos considerar a essência.

A essência de uma entidade de software é uma construção de conceitos interligados: conjuntos de dados, relacionamentos entre itens de dados, algoritmos e invocações de funções. Essa essência é abstrata, no sentido de que a construção conceitual é a mesma sob muitas representações diferentes. No entanto, é altamente preciso e ricamente detalhado.

I acredito que a parte difícil de construir software seja a especificação, projeto e teste dessa construção conceitual, não o trabalho de representá-la e testar a fidelidade da representação. Ainda cometemos erros de sintaxe, com certeza; mas eles são confusos em comparação com os erros conceituais na maioria dos sistemas.

Se isso for verdade, construir software sempre será difícil. Não há nenhuma bala de prata inerentemente.

Vamos considerar as propriedades inerentes dessa essência irredutível dos sistemas de software modernos: complexidade, conformidade, mutabilidade e invisibilidade.

Complexidade. As entidades de software são mais complexas para seu tamanho do que talvez qualquer outra construção humana, porque não há duas partes iguais (pelo menos acima do nível de instrução). Se forem, transformamos as duas partes semelhantes em uma, uma sub-rotina, aberta ou fechada. Nesse aspecto, os sistemas de software diferem profundamente de computadores, edifícios ou automóveis, onde abundam os elementos repetidos.

Os próprios computadores digitais são mais complexos do que a maioria das coisas que as pessoas constroem; eles têm um grande número de estados. Isso torna difícil concebê-los, descrevê-los e testá-los. Os sistemas de software têm ordens de magnitude mais estados do que os computadores.

Da mesma forma, a ampliação de uma entidade de software não é meramente uma repetição dos mesmos elementos em tamanho maior; é necessariamente um aumento no número de elementos diferentes. Na maioria dos casos, os elementos interagem entre si de alguma forma não linear e a complexidade do todo aumenta muito mais do que linearmente.

A complexidade do software é uma propriedade essencial, não acidental. Conseqüentemente, as descrições de uma entidade de software que abstraem sua complexidade freqüentemente abstraem sua essência. A matemática e as ciências físicas fizeram grandes avanços durante três séculos, construindo modelos simplificados de fenômenos complexos, derivando propriedades dos modelos e verificando essas propriedades experimentalmente. Isso funcionou porque as complexidades ignoradas nos modelos não eram as propriedades essenciais dos fenômenos. Não funciona quando as complexidades são a essência.

Muitos dos problemas clássicos de desenvolvimento de produtos de software derivam dessa complexidade essencial e seu não linear aumenta com o tamanho. Da complexidade vem a dificuldade de comunicação entre os membros da equipe, o que leva a falhas no produto, estouro de custos, atrasos no cronograma. Da complexidade vem a dificuldade de enumerar, muito menos de compreender, todos os estados possíveis do programa, e daí a insegurança. Da complexidade das funções vem a dificuldade de invocar essas funções, o que torna os programas difíceis de usar. Da complexidade da estrutura vem a dificuldade de estender programas para novas funções sem criar efeitos colaterais. Da complexidade da estrutura vêm os estados não visualizados que constituem alçapões de segurança.

Não apenas os problemas técnicos, mas também os problemas de gerenciamento vêm da complexidade. Essa complexidade torna mais

visão difícil, impedindo assim a integridade conceitual. Isso torna difícil encontrar e controlar todas as pontas soltas. Isso cria uma enorme carga de aprendizado e compreensão que torna a rotatividade de pessoal um desastre.

Conformidade. O pessoal de software não está sozinho ao enfrentar a complexidade. A física lida com objetos terrivelmente complexos, mesmo no nível de partícula "fundamental". O físico trabalha, entretanto, em uma fé firme de que existem princípios unificadores a serem encontrados, seja em quarks ou em teorias de campo unificado. Einstein argumentou repetidamente que deve haver explicações simplificadas da natureza, porque Deus não é caprichoso ou arbitrário.

Essa fé não conforta o engenheiro de software. Muito da complexidade hemust master é uma complexidade arbitrária, forçada sem rima ou razão pelas muitas instituições e sistemas humanos aos quais suas interfaces devem se conformar. Eles diferem de interface para interface e de vez em quando, não por causa da necessidade, mas apenas porque foram projetados por pessoas diferentes, em vez de por Deus.

Em muitos casos, o software deve estar em conformidade porque ele entrou em cena mais recentemente. Em outros, deve estar em conformidade porque é percebido como o mais adaptável. Mas em todos os casos, muita complexidade vem da conformação para outras interfaces; isso não pode ser simplificado por qualquer redesenho do software sozinho.

Mutabilidade. A entidade de software está constantemente sujeita a pressões de mudança. É claro que o mesmo acontece com edifícios, carros, computadores. Mas as coisas manufaturadas raramente são alteradas após a manufatura; eles são substituídos por modelos posteriores ou alterações essenciais são incorporadas em cópias posteriores de número de série do mesmo projeto básico. As chamadas de retorno de automóveis são realmente muito raras; mudanças de campo de computadores um pouco menos. Ambos são muito menos frequentes do que modificações no software em campo.

Em parte, isso ocorre porque o software em um sistema incorpora sua função, e a função é a parte que mais sente as pressões da mudança. Em parte é porque o software pode ser alterado

mais facilmente - é puro pensamento, infinitamente maleável. Os edifícios, de fato, são alterados, mas os altos custos da mudança, compreendidos por todos, servem para amortecer os caprichos dos que mudam.

Todos os softwares de sucesso são alterados. Dois processos estão em ação. À medida que um produto de software é considerado útil, as pessoas o experimentam em novos casos, no limite ou além do domínio original. As pressões para a função estendida vêm principalmente de usuários que gostam da função básica e inventam novos usos para ela.

Em segundo lugar, o software de sucesso também sobrevive além da vida normal do veículo da máquina para o qual foi escrito pela primeira vez. Se não novos computadores, pelo menos novos discos, novos monitores, novas impressoras surgem; e o software deve estar em conformidade com seus novos veículos de oportunidade.

Em suma, o produto de software está inserido em uma matriz cultural de aplicativos, usuários, leis e veículos de máquinas. Todos eles mudam continuamente e suas alterações forçam inexoravelmente a mudança no produto de software.

Invisibilidade. O software é invisível e não visível. Abstrações geométricas são ferramentas poderosas. A planta baixa de um edifício ajuda o arquiteto e o cliente a avaliar espaços, fluxos de tráfego e vistas. As contradições tornam-se óbvias, as omissões podem ser detectadas. Desenhos em escala de peças mecânicas e modelos de moléculas em forma de palito, embora abstrações, têm o mesmo propósito. Uma realidade geométrica é capturada em uma abstração geométrica.

A realidade do software não está inherentemente embutida no espaço. Portanto, não há uma representação geométrica pronta da mesma forma que a terra tem mapas, os chips de silício têm diagramas, os computadores têm esquemas de conectividade. Assim que tentamos diagramar a estrutura do software, descobrimos que ela constitui não um, mas vários gráficos direcionados gerais, sobrepostos uns aos outros. Os vários gráficos podem representar o fluxo de controle, o fluxo de dados, padrões de dependência, sequência de tempo, relações de espaço de nomes. Geralmente, eles não são planos, muito menos hierárquicos. Na verdade, uma das maneiras de estabelecer controle conceitual sobre tal estrutura é forçar o corte do link até que um ou

mais gráficos tornam-se hierárquicos.²

Apesar do progresso em restringir e simplificar as estruturas do software, elas permanecem inherentemente invisíveis, privando assim a mente de algumas de suas ferramentas conceituais mais poderosas. Essa falta não apenas impede o processo de design dentro de uma mente, mas também prejudica gravemente a comunicação entre as mentes.

Avanços passados resolveram dificuldades acidentais

Se examinarmos as três etapas da tecnologia de software que foram mais frutíferas no passado, descobriremos que cada uma atacou uma dificuldade principal diferente na construção de software, mas essas foram as dificuldades acidentais, não as essenciais. Também podemos ver os limites naturais para a extração de cada um desses ataques.

Linguagens de alto nível. Certamente, o golpe mais poderoso para produtividade, confiabilidade e simplicidade de software tem sido o uso progressivo de linguagens de alto nível para programação. A maioria dos observadores atribui a esse desenvolvimento pelo menos um fator de cinco em produtividade e ganhos concomitantes em confiabilidade, simplicidade e compreensibilidade.

O que uma linguagem de alto nível realiza? Ele libera um programa de grande parte de sua complexidade acidental. Um programa abstrato consiste em construções conceituais: operações, tipos de dados, sequências e comunicação. O programa de máquina concreto se preocupa com bits, registros, condições, ramificações, canais, discos e outros. Na medida em que a linguagem de alto nível incorpora as construções desejadas no programa abstrato e evita todas as inferiores / elimina todo um nível de complexidade que nunca foi inherent ao programa.

O máximo que uma linguagem de alto nível pode fazer é fornecer todas as construções que o programador imagina no programa abstrato. Para ter certeza, o nível de nossa sofisticação em pensar sobre estruturas de dados, tipos de dados e operações está aumentando continuamente, mas a uma taxa cada vez menor. E o desenvolvimento da linguagem se aproxima cada vez mais da sofisticação dos usuários.

Além disso, em algum ponto a elaboração de um Ian- de alto nível

a linguagem torna-se um fardo que aumenta, e não reduz, a tarefa intelectual do usuário que raramente usa as construções esotéricas.

Compartilhamento de tempo. A maioria dos observadores credita o time-sharing a uma grande melhoria na produtividade dos programadores e na qualidade de seus produtos, embora não tão grande quanto a proporcionada por linguagens de alto nível.

O compartilhamento de tempo ataca uma dificuldade distintamente diferente. O compartilhamento de tempo preserva o imediatismo e, portanto, nos permite manter uma visão geral da complexidade. A lenta recuperação da programação em lote significa que inevitavelmente esquecemos as minúcias, senão o próprio impulso, em que estávamos pensando quando paramos de programar e pedimos a compilação e a execução. Essa interrupção da consciência custa muito tempo, pois devemos nos refrescar. O efeito mais sério pode muito bem ser a decadência de tudo o que está acontecendo em um sistema complexo.

A recuperação lenta, como as complexidades da linguagem de máquina, é uma dificuldade acidental, e não essencial, do processo de software. Os limites da contribuição do time-sharing derivam diretamente. O principal efeito é reduzir o tempo de resposta do sistema. À medida que vai para zero, em algum ponto ele ultrapassa o limite humano de perceptibilidade, cerca de 100 milissegundos. Além disso, nenhum benefício deve ser esperado.

Ambientes de programação unificados. Unix e Interlisp, os primeiros ambientes de programação integrada a serem amplamente utilizados, são percebidos como tendo uma produtividade aprimorada por fatores integrais. Porque?

Eles atacam as dificuldades acidentais de usar programas *juntos*, fornecendo bibliotecas integradas, formatos de arquivo unificados e canais e filtros. Como resultado, as estruturas conceituais que, em princípio, sempre poderiam chamar, alimentar e usar umas às outras podem, de fato, fazer isso facilmente na prática.

Essa descoberta, por sua vez, estimulou o desenvolvimento de bancos de ferramentas inteiros, uma vez que cada nova ferramenta poderia ser aplicada a qualquer programa usando os formatos padrão.

Por causa desses sucessos, os ambientes estão sujeitos a

grande parte da pesquisa de engenharia de software de hoje. Veremos suas promessas e limitações na próxima seção.

Esperanças pelo Silver

Agora, consideremos os desenvolvimentos técnicos mais frequentemente avançados como potenciais balas de prata. Que problemas eles abordam? São os problemas da essência ou são resquícios de nossas dificuldades accidentais? Oferecem avanços revolucionários ou incrementais?

Ada e outros avanços da linguagem de alto nível. Um dos desenvolvimentos recentes mais elogiados é a linguagem de programação Ada, uma linguagem de uso geral e alto nível da década de 1980. Ada, de fato, não apenas reflete melhorias evolutivas nos conceitos de linguagem, mas incorpora recursos para encorajar o design moderno e os conceitos de modularização. Talvez a filosofia Ada seja mais um avanço do que a linguagem Ada, pois é a filosofia da modularização, dos tipos de dados abstratos, da estruturação hierárquica. Ada talvez seja muito rica, o produto natural do processo pelo qual os requisitos foram colocados em seu design. Isso não é fatal, pois o subconjunto de vocabulários de trabalho pode resolver o problema de aprendizado, e os avanços do hardware nos darão o MIPS barato para pagar pelos custos de compilação. Avançar na estruturação de sistemas de software é, de fato, um uso muito bom para o aumento de MIPS que nosso dinheiro comprará. Os sistemas operacionais, altamente criticados na década de 1960 por seus custos de memória e ciclo, provaram ser uma excelente forma de usar alguns dos MIPS e bytes de memória baratos do surto de hardware anterior.

No entanto, Ada não provará ser a bala de prata que mata o monstro da produtividade de software. Afinal, é apenas mais uma linguagem de alto nível, e a maior recompensa dessas linguagens veio da primeira transição, das complexidades accidentais da máquina para o enunciado mais abstrato de soluções passo a passo. Uma vez que esses acidentes tenham sido removidos, os restantes são menores e a recompensa de sua remoção certamente será menor.

Prevejo que daqui a uma década, quando a eficácia de Ada for avaliada, verá que fez uma diferença substancial, mas não por causa de qualquer característica específica da linguagem, nem mesmo por causa de todos eles combinados. Nem os novos ambientes Ada provarão ser a causa das melhorias. A maior contribuição de Ada será que a mudança para ele ocasionou o treinamento de programadores em técnicas modernas de design de software.

Programação orientada a objetos. Muitos estudantes da arte defendem tem mais esperança para a programação orientada a objetos do que para qualquer outro modismo técnico da época.³ Eu estou entre eles. Mark Sherman, de Dartmouth, observa que devemos ter o cuidado de distinguir duas ideias separadas que têm esse nome: tipos de dados abstratos e tipos hierárquicos, também chamados de *Aulas*. O conceito do tipo de dado abstrato é que o tipo de um objeto deve ser definido por um nome, um conjunto de valores adequados e um conjunto de operações adequadas, em vez de sua estrutura de armazenamento, que deve ser ocultada. Exemplos são pacotes Ada (com tipos privados) ou módulos Modula.

Os tipos hierárquicos, como as classes do Simula-67, permitem a definição de interfaces gerais que podem ser ainda mais refinadas fornecendo tipos subordinados. Os dois conceitos são ortogonais - pode haver hierarquias sem ocultar e ocultar sem hierarquias. Ambos os conceitos representam avanços reais na arte de construção de software.

Cada um remove mais uma dificuldade acidental do processo, permitindo ao designer expressar a essência de seu design sem ter que expressar grandes quantidades de material sintático que não adiciona nenhum novo conteúdo de informação. Para tipos abstratos e tipos hierárquicos, o resultado é remover um tipo de dificuldade acidental de ordem superior e permitir uma expressão de design de ordem superior.

No entanto, tais avanços não podem fazer mais do que remover todas as dificuldades acidentais da expressão do design. A complexidade do próprio design é essencial; e tal

os ataques não fazem nenhuma mudança nisso. Um ganho de ordem de magnitude pode ser obtido pela programação orientada a objetos somente se a vegetação rasteira desnecessária de especificação de tipo que permanece hoje em nossa linguagem de programação for responsável por dezenove avos do trabalho envolvido no projeto de um produto de programa. Eu duvido.

Inteligência artificial. Muitas pessoas esperam que os avanços em inteligência artificial proporcionem o avanço revolucionário que proporcionará ganhos de ordem de magnitude em produtividade e qualidade de software.⁴ 1 não. Para ver por quê, devemos dissecar o que significa "inteligência artificial" e então ver como ela se aplica.

Parnas esclareceu o caos terminológico:

Duas definições bastante diferentes de IA são comumente usadas hoje. AI-1: O uso de computadores para resolver problemas que antes só podiam ser resolvidos com a aplicação da inteligência humana. AI2: O uso de um conjunto específico de técnicas de programação, conhecido como programação heurística ou baseada em regras. Nesta abordagem, especialistas humanos são estudados para determinar quais heurísticas ou regras práticas eles usam na resolução de problemas. . . . O programa é projetado para resolver um problema da maneira que os humanos parecem resolvê-lo.

*A primeira definição tem um significado deslizante. . . . Algo pode se encaixar na definição de AI-1 hoje, mas, uma vez que vermos como o programa funciona e entender o problema, não pensaremos mais nele como AI. . . . Infelizmente, não consigo identificar um corpo de tecnologia exclusivo para este campo. . . . A maior parte do trabalho é específico para o problema, e alguma abstração ou criatividade é necessária para ver como transferi-lo.*⁵

Eu concordo totalmente com essa crítica. As técnicas utilizadas para reconhecimento de fala parecem ter pouco em comum com aquelas utilizadas para reconhecimento de imagem, e ambas são diferentes daquelas utilizadas em sistemas especialistas. Tenho dificuldade em ver como o reconhecimento de imagem, por exemplo, fará qualquer diferença apreciável na prática de programação. O mesmo é verdadeiro para o reconhecimento de fala

ção O difícil de criar software é decidir o que dizer, não dizer.
Nenhuma facilitação de expressão pode dar mais do que ganhos marginais.

A tecnologia de sistemas especializados, AI-2, merece uma seção própria.

Sistemas especializados. A parte mais avançada da arte da inteligência artificial, e a mais amplamente aplicada, é a tecnologia de construção de sistemas especialistas. Muitos cientistas de software estão trabalhando arduamente para aplicar essa tecnologia ao ambiente de construção de software.⁵ Qual é o conceito e quais são as perspectivas?

Um sistema especialista é um programa que contém um mecanismo de inferência generalizada e uma base de regra, projetado para obter dados de entrada e suposições e explorar as consequências lógicas por meio das inferências deriváveis da base de regra, produzindo conclusões e conselhos e oferecendo a explicação de seus resultados reconstituindo seu raciocínio para o usuário. Os mecanismos de inferência normalmente podem lidar com dados e regras difusas ou probabilísticas, além da lógica puramente determinística.

Esses sistemas oferecem algumas vantagens claras sobre os algoritmos programados para chegar às mesmas soluções para os mesmos problemas:

- A tecnologia do mecanismo de inferência é desenvolvida de forma independente do aplicativo e, em seguida, aplicada a muitos usos. Pode-se justificar muito mais esforço nos motores de inferência. Na verdade, essa tecnologia está bem avançada.
- As partes mutáveis dos materiais peculiares da aplicação são codificadas na base de regra de maneira uniforme e as ferramentas são fornecidas para desenvolver, alterar, testar e documentar a base de regra. Isso regulariza grande parte da complexidade do próprio aplicativo.

Edward Feigenbaum diz que o poder de tais sistemas não vem de mecanismos de inferência cada vez mais sofisticados, mas sim de bases de conhecimento cada vez mais ricas que refletem o mundo real com mais precisão. Acredito que o avanço mais importante oferecido pela tecnologia é a separação do aplicativo.

plexidade do próprio programa.

Como isso pode ser aplicado à tarefa de software? De muitas maneiras: sugerindo regras de interface, aconselhando sobre estratégias de teste, lembrando as frequências de tipo de bug, oferecendo dicas de otimização, etc.

Considere um consultor de teste imaginário, por exemplo. Em sua forma mais rudimentar, o sistema especialista de diagnóstico é muito semelhante a uma lista de verificação do piloto, fundamentalmente oferecendo sugestões quanto às possíveis causas de dificuldade. À medida que a base de regras é desenvolvida, as sugestões se tornam mais específicas, levando em consideração de forma mais sofisticada os sintomas de problemas relatados. Pode-se visualizar um assistente de depuração que oferece sugestões muito generalizadas no início, mas à medida que mais e mais a estrutura do sistema é incorporada na base de regras, torna-se cada vez mais particular nas hipóteses que gera e nos testes que recomenda. Tal sistema especialista pode se afastar mais radicalmente dos convencionais, pois sua base de regras provavelmente deve ser modularizada hierarquicamente da mesma forma que o produto de software correspondente, de modo que, à medida que o produto é modificado modularmente,

O trabalho necessário para gerar as regras de diagnóstico é um trabalho que deverá ser realizado de qualquer maneira na geração do conjunto de casos de teste para os módulos e para o sistema. Se for feito de maneira geral adequada, com uma estrutura uniforme de regras e um bom mecanismo de inferência disponível, pode realmente reduzir o trabalho total de geração de casos de teste de preparação, bem como ajudar na manutenção vitalícia e nos testes de modificação. Da mesma forma, podemos postular outros orientadores - provavelmente muitos dos comandos provavelmente simples - para as outras partes da tarefa de construção de software.

Muitas dificuldades impedem a realização precoce de consultores especializados úteis para o desenvolvedor do programa. Uma parte crucial de nosso cenário imaginário é o desenvolvimento de maneiras fáceis de ir da especificação da estrutura do programa à geração automática ou semiautomática de regras de diagnóstico. Ainda mais difícil e importante é a dupla tarefa de aquisição de conhecimento:

encontrar especialistas articulados e auto-analíticos que sabem *porque* eles fazem coisas; e desenvolver técnicas eficientes para extrair o que eles sabem e destilá-lo em bases de regras. O pré-requisito essencial para construir um sistema especialista é ter um especialista.

A contribuição mais poderosa dos sistemas especialistas será certamente colocar a serviço do programador inexperiente a experiência e sabedoria acumulada dos melhores programadores. Esta não é uma contribuição pequena. A lacuna entre a melhor prática de engenharia de software e a prática média é muito grande - talvez maior do que em qualquer outra disciplina de engenharia. Uma ferramenta que dissemine boas práticas seria importante.

Programação "automática". Por quase 40 anos, as pessoas anteciparam e escreveram sobre "programação automática", a geração de um programa para resolver um problema a partir de uma declaração das especificações do problema. Algumas pessoas hoje escrevem como se esperassesem que essa tecnologia proporcionasse o próximo avanço.⁷

Parnas implica que o termo é usado para glamour e não para conteúdo semântico, afirmando,

Em suma, a programação automática sempre foi um eufemismo para programar com uma linguagem de nível superior do que a que estava atualmente disponível para o programador.³

Ele argumenta, em essência, que na maioria dos casos é o método de solução, não o problema, cuja especificação deve ser fornecida.

Exceções podem ser encontradas. A técnica de construção de geradores é muito poderosa e é rotineiramente usada com grande vantagem em programas de classificação. Alguns sistemas de integração de equações diferenciais também permitiram a especificação direta do problema. O sistema avaliou os parâmetros, escolheu em uma biblioteca de métodos de solução e gerou os programas.

Esses aplicativos têm propriedades muito favoráveis:

- Os problemas são facilmente caracterizados por relativamente poucos pacientes parâmetros.
- Existem muitos métodos conhecidos de solução para fornecer uma biblioteca de alternativas.

- A análise extensiva levou a regras explícitas para selecionar técnicas de solução, dados os parâmetros do problema.

É difícil ver como essas técnicas se generalizam para o mundo mais amplo do sistema de software comum, onde casos com propriedades tão simples são a exceção. É difícil até imaginar como esse avanço na generalização poderia ocorrer.

Programação gráfica. Uma matéria favorita para Ph.D. dissertações em engenharia de software é a programação gráfica ou visual, a aplicação da computação gráfica ao design de software.⁹

Às vezes, a promessa de tal abordagem é postulada a partir da analogia com o design do chip VLSI, onde a computação gráfica desempenha um papel muito frutífero. Às vezes, a abordagem é justificada considerando os fluxogramas como o meio de design de programa ideal e fornecendo recursos poderosos para construí-los.

Nada nem mesmo convincente, muito menos excitante, emergiu de tais esforços. Estou convencido de que nada acontecerá.

Em primeiro lugar, como argumentei em outro lugar, o fluxograma é uma abstração muito pobre da estrutura de software.¹⁰ Na verdade, é melhor visto como a tentativa de Burks, von Neumann e Goldstine de fornecer uma linguagem de controle de alto nível desesperadamente necessária para o computador proposto. Na forma lamentável, com várias páginas e em caixa de conexão, para a qual o fluxograma foi elaborado hoje, ele provou ser essencialmente inútil como ferramenta de design - os programadores desenham fluxogramas depois, não antes, de escrever os programas que descrevem.

Em segundo lugar, as telas de hoje são muito pequenas, em pixels, para mostrar tanto o escopo quanto a resolução de qualquer diagrama de software sério e detalhado. A chamada "metáfora de mesa" da estação de trabalho de hoje é, em vez disso, uma metáfora de "assento de avião". Qualquer um que embaralhou uma pilha de papéis enquanto estava sentado na carruagem entre dois passageiros corpulentos reconhecerá a diferença - só se pode ver muito poucas coisas ao mesmo tempo. A verdadeira área de trabalho oferece uma visão geral e acesso aleatório a várias páginas. Além disso, quando os ataques de criatividade são fortes, mais de um programador ou escritor abandonou a área de trabalho por

o andar mais espaçoso. A tecnologia de hardware terá que avançar substancialmente antes que o escopo de nossos escopos seja suficiente para a tarefa de design de software.

Mais fundamentalmente, como argumentei acima, o software é muito difícil de visualizar. Quer façamos diagramas de fluxo de controle, aninhamento de escopo variável, referências cruzadas de variáveis, fluxo de dados, estruturas de dados hierárquicas ou o que quer que seja, sentimos apenas uma dimensão do elefante de software intrincadamente interligado. Se sobrepormos todos os diagramas gerados pelas muitas visualizações relevantes, será difícil extrair qualquer visão geral global. A analogia do VLSI é fundamentalmente enganosa - o design de um chip é um objeto bidimensional em camadas cuja geometria reflete sua essência. Um sistema de software não é.

Verificação do programa. Grande parte do esforço da programação moderna é direcionada ao teste e reparo de bugs. Existe talvez uma solução mágica para eliminar os erros na origem, na fase de projeto do sistema? A produtividade e a confiabilidade do produto podem ser radicalmente aprimoradas seguindo a estratégia profundamente diferente de provar que os projetos estão corretos antes que o imenso esforço seja feito para implementá-los e testá-los?

Não acredito que encontraremos a magia aqui. A verificação do programa é um conceito muito poderoso e será muito importante para coisas como kernels de sistema operacional seguros. A tecnologia não promete, porém, economia de mão de obra. As verificações são tão trabalhosas que apenas alguns programas substanciais foram verificados.

A verificação do programa não significa programas à prova de erros. Não há mágica aqui também. As provas matemáticas também podem apresentar falhas. Portanto, embora a verificação possa reduzir a carga de teste do programa, ela não pode eliminá-la.

Mais seriamente, mesmo a verificação perfeita do programa só pode estabelecer que um programa atende às suas especificações. A parte mais difícil da tarefa de software é chegar a uma especificação completa e consistente, e muito da essência da construção de um programa é, na verdade, a depuração da especificação.

Ambientes e ferramentas. Quanto ganho a mais pode ser esperado das pesquisas explosivas em melhores ambientes de programação? A reação instintiva de alguém é que os problemas da grande recompensa foram os primeiros atacados e foram resolvidos: sistemas de arquivos hierárquicos, formatos de arquivo uniformes para ter interfaces de programa uniformes e ferramentas generalizadas. Editores inteligentes específicos de linguagem são desenvolvimentos ainda não amplamente usados na prática, mas o máximo que eles prometem é a liberdade de erros sintáticos e erros semânticos simples.

Talvez o maior ganho a ser realizado no ambiente de programação seja o uso de sistemas de banco de dados integrados para manter o controle da miríade de detalhes que devem ser recuperados com precisão pelo programador individual e mantidos atualizados em um grupo de colaboradores em um único sistema.

Certamente este trabalho vale a pena e certamente renderá alguns frutos em produtividade e confiabilidade. Mas, por sua própria natureza, o retorno de agora em diante deve ser marginal.

Estações de trabalho. Que ganhos podem ser esperados para a arte do software com o aumento rápido e seguro na capacidade de potência e memória de cada estação de trabalho? Bem, quantos MIPS se pode usar com sucesso? A composição e edição de programas e documentos são totalmente suportadas pelas velocidades atuais. Compilar pode ser um grande impulso, mas um fator de 10 na velocidade da máquina certamente deixaria o tempo de reflexão a atividade dominante na época do programador. Na verdade, parece que sim agora.

Estações de trabalho mais potentes certamente serão bem-vindas. Melhorias mágicas deles não podemos esperar.

Ataques promissores à essência conceitual

Mesmo que nenhum avanço tecnológico prometa dar o tipo de resultados mágicos com os quais estamos tão familiarizados na área de hardware, há uma abundância de bons trabalhos em andamento agora e a promessa de progresso constante, embora nada espetacular.

Todos os ataques tecnológicos aos acidentes do soft-

processo de armazenamento são fundamentalmente limitados pela equação de produtividade:

$$\text{Time of task} = \sum_i (\text{Frequency})_i \times (\text{Time})_i$$

Se, como acredito, os componentes conceituais da tarefa agora estão consumindo a maior parte do tempo, nenhuma atividade nos componentes da tarefa que sejam meramente a expressão dos conceitos pode gerar grandes ganhos de produtividade.

Portanto, devemos considerar os ataques que abordam a essência do problema de software, a formulação dessas estruturas conceituais complexas. Felizmente, alguns deles são muito promissores.

Compre versus construa. A solução mais radical possível para construir software é simplesmente não construí-lo.

A cada dia isso se torna mais fácil, à medida que mais e mais fornecedores oferecem mais e melhores produtos de software para uma variedade estonteante de aplicativos. Embora nós, engenheiros de software, trabalhemos na metodologia de produção, a revolução do computador pessoal criou não um, mas muitos mercados de massa para software. Cada banca de jornal traz revistas mensais que, classificadas por tipo de máquina, anunciam e analisam dezenas de produtos a preços de alguns dólares a algumas centenas de dólares. Fontes mais especializadas oferecem produtos muito poderosos para estações de trabalho e outros mercados Unix. Até mesmo ferramentas de software e ambientes podem ser adquiridos imediatamente. Eu propus em outro lugar um mercado para módulos individuais.

Qualquer produto desse tipo é mais barato comprar do que construir de novo. Mesmo a um custo de \$ 100.000, um software comprado está custando apenas cerca de um programador por ano. E a entrega é imediata! Imediato pelo menos para produtos que realmente existem, produtos cujo desenvolvedor pode encaminhar o cliente potencial para um usuário feliz. Além disso, esses produtos tendem a ser muito mais bem documentados e mantidos de alguma forma melhor do que os softwares desenvolvidos internamente.

O desenvolvimento do mercado de massa é, acredito, a tendência de longo prazo mais profunda na engenharia de software. O custo de

software sempre foi custo de desenvolvimento, não custo de replicação. Compartilhar esse custo entre mesmo alguns usuários corta radicalmente o custo por usuário. Outra maneira de ver isso é que o uso de n cópias de um sistema de software efetivamente multiplica a produtividade de seus desenvolvedores por n . Isso é um aumento da produtividade da disciplina e da nação.

A questão principal, é claro, é a aplicabilidade. Posso usar um pacote disponível no mercado para fazer minha tarefa? Uma coisa surpreendente aconteceu aqui. Durante as décadas de 1950 e 1960, estudo após estudo mostrou que os usuários não usariam pacotes prontos para uso para folha de pagamento, controle de estoque, contas a receber, etc. Os requisitos eram muito especializados, a variação caso a caso muito alta. Durante a década de 1980, encontramos esses pacotes em alta demanda e uso generalizado. O que mudou?

Na verdade, não os pacotes. Eles podem ser um pouco mais generalizados e personalizáveis do que antes, mas não muito. Nem mesmo os aplicativos. No mínimo, as necessidades empresariais e científicas de hoje são mais diversificadas, mais complicadas do que as de 20 anos atrás.

A grande mudança foi na relação custo de hardware / software. O comprador de uma máquina de US \$ 2 milhões em 1960 sentiu que poderia pagar US \$ 250.000 a mais por um programa de folha de pagamento personalizado, que escorregou fácil e sem interrupções para o ambiente social hostil ao computador. Hoje, os compradores de máquinas de escritório de US \$ 50.000 não podem pagar programas de folha de pagamento personalizados; para que adaptem seus procedimentos de folha de pagamento aos pacotes disponíveis. Os computadores são agora tão comuns, se ainda não tão amados, que as adaptações são aceitas como algo natural.

Existem exceções dramáticas ao meu argumento de que a generalização dos pacotes de software mudou pouco ao longo dos anos: planilhas eletrônicas e sistemas de banco de dados simples. Essas ferramentas poderosas, tão óbvias em retrospecto e ainda assim surgindo tão tarde, se prestam a uma miríade de usos, alguns bastante pouco ortodoxos. Artigos e até livros agora abundam sobre como lidar com tarefas inesperadas com a planilha. Um grande número de aplicativos que anteriormente teriam sido escritos como programas personalizados

gramas em Cobol ou Report Program Generator agora são rotineiramente feitas com essas ferramentas.

Muitos usuários agora operam seus próprios computadores dia após dia em diversos aplicativos, sem nunca escrever um programa. De fato, muitos desses usuários não podem escrever novos programas para suas máquinas, mas, mesmo assim, são adeptos a resolver novos problemas com eles.

Acredito que a estratégia de produtividade de software mais poderosa para muitas organizações hoje é equipar os trabalhadores intelectuais da computação na linha de fogo com computadores pessoais e bons programas generalizados de escrita, desenho, arquivo e planilha, e soltá-los. A mesma estratégia, com pacotes matemáticos e estatísticos generalizados e alguns recursos de programação simples, também funcionará para centenas de cientistas de laboratório.

Refinamento de requisitos e prototipagem rápida. A parte mais difícil de construir um sistema de software é decidir exatamente o que construir. Nenhuma outra parte do trabalho conceitual é tão difícil quanto estabelecer os requisitos técnicos detalhados, incluindo todas as interfaces para pessoas, máquinas e outros sistemas de software. Nenhuma outra parte do trabalho prejudica tanto o sistema resultante se feito de maneira errada. Nenhuma outra parte é mais difícil de corrigir posteriormente.

Portanto, a função mais importante que os criadores de software desempenham para seus clientes é a extração iterativa e o refinamento dos requisitos do produto. Pois a verdade é que os clientes não sabem o que querem. Eles geralmente não sabem quais perguntas devem ser respondidas e quase nunca pensaram no problema nos detalhes que devem ser especificados. Mesmo a resposta simples - "Faça o novo sistema de software funcionar como nosso antigo sistema manual de processamento de informações" - é na verdade muito simples. Os clientes nunca querem exatamente isso. Além disso, sistemas de software complexos são coisas que agem, que se movem, que funcionam. A dinâmica dessa ação é difícil de imaginar. Portanto, no planejamento de qualquer atividade de software, é necessário permitir uma iteração extensiva entre o cliente e o designer como parte da definição do sistema.

Eu daria um passo além e afirmaria que é realmente impossível para os clientes, mesmo aqueles que trabalham com engenheiros de software, especificar completa, precisa e corretamente os requisitos exatos de um produto de software moderno antes de construir e experimentar algumas versões do produto que eles estão especificando.

Portanto, um dos mais promissores dos esforços tecnológicos atuais, e que ataca a essência, e não os acidentes, do problema de software, é o desenvolvimento de abordagens e ferramentas para prototipagem rápida de sistemas como parte da especificação iterativa de requisitos.

Um sistema de software protótipo é aquele que simula as interfaces importantes e executa as funções principais do sistema pretendido, embora não seja necessariamente limitado pela mesma velocidade de hardware, tamanho ou restrições de custo. Os protótipos normalmente executam as tarefas principais do aplicativo, mas não fazem nenhuma tentativa de lidar com as exceções, respondem corretamente a entradas inválidas, abortam de forma limpa, etc. O objetivo do protótipo é tornar real a estrutura conceitual especificada, para que o cliente possa testá-la quanto à consistência e usabilidade.

Muitos dos procedimentos atuais de aquisição de software baseiam-se na suposição de que se pode especificar um sistema satisfatório com antecedência, obter propostas para sua construção, mandar construí-lo e instalá-lo. Acho que essa suposição está fundamentalmente errada e que muitos problemas de aquisição de software surgem dessa falácia. Conseqüentemente, eles não podem ser corrigidos sem uma revisão fundamental, que forneça o desenvolvimento iterativo e a especificação de protótipos e produtos.

Desenvolvimento incremental — Crescer, não construir, software. Ainda me lembro do choque que senti em 1958, quando ouvi pela primeira vez um amigo falar sobre *construindo um programa*, em oposição a *escrita* 1. Num piscar de olhos, ampliei toda a minha visão do processo de software. A mudança da metáfora foi poderosa e precisa. Hoje entendemos como a construção de software é semelhante a outros processos de construção e usamos livremente outros elementos da metáfora, como *especificações, montagem de componentes, e andaime*.

A metáfora da construção perdeu sua utilidade. É hora de mudar novamente. Se, como acredito, as estruturas conceituais que construímos hoje são muito complicadas para serem especificadas com precisão com antecedência, e muito complexas para serem construídas sem falhas, então devemos adotar uma abordagem radicalmente diferente.

Vamos nos voltar para a natureza e estudar a complexidade das coisas vivas, em vez de apenas as obras mortas do homem. Aqui encontramos construções cujas complexidades nos deixam maravilhados. O cérebro sozinho é intrincado além do mapeamento, poderoso além da imitação, rico em diversidade, autoprotetor e auto-renovador. O segredo é que ele cresceu, não foi construído.

Assim deve ser com nossos sistemas de software. Alguns anos atrás, Harlan Mills propôs que qualquer sistema de software deveria ser desenvolvido por desenvolvimento incremental.^{onze} Ou seja, o sistema deve primeiro ser feito para funcionar, mesmo que ele não faça nada útil, exceto chamar o conjunto apropriado de subprogramas fictícios. Então, pouco a pouco, ele é desenvolvido, com os subprogramas, por sua vez, sendo desenvolvidos em ações ou chamadas para esvaziar tocos no nível abaixo.

Tenho visto os resultados mais dramáticos desde que comecei a recomendar essa técnica aos criadores de projetos em minha aula de laboratório de engenharia de software. Nada na última década mudou tão radicalmente minha prática ou sua eficácia. A abordagem requer um design de cima para baixo, pois é um crescimento de cima para baixo do software. Ele permite um retrocesso fácil. Ele se presta aos primeiros protótipos. Cada função adicionada e nova provisão para dados ou circunstâncias mais complexas crescem organicamente do que já existe.

Os efeitos do moral são surpreendentes. O entusiasmo salta quando há um sistema em execução, mesmo que seja simples. Os esforços redobram quando a primeira imagem de um novo sistema de software gráfico aparece na tela, mesmo que seja apenas um retângulo. Sempre se tem, em cada etapa do processo, um sistema de trabalho. Eu acho que as equipes podem *crescer* entidades muito mais complexas em quatro meses do que podem *construir*.

Os mesmos benefícios podem ser obtidos em projetos grandes e pequenos.¹²

Grandes designers. A questão central de como melhorar o software art centra-se, como sempre, nas pessoas.

Podemos obter bons projetos seguindo boas práticas em vez de práticas ruins. Boas práticas de design podem ser ensinadas. Os programadores estão entre a parte mais inteligente da população, por isso podem aprender boas práticas. Portanto, um grande impulso nos Estados Unidos é a promulgação de boas práticas modernas. Novos currículos, nova literatura, novas organizações como o Software Engineering Institute, tudo surgiu para elevar o nível de nossa prática de ruim para bom. Isso é inteiramente ~~proper~~.

No entanto, não acredito que possamos dar o próximo passo para cima da mesma forma. Enquanto a diferença entre projetos conceituais ruins e bons pode estar na solidez do método de design, a diferença entre projetos bons e excelentes certamente não está. Grandes designs vêm de grandes designers. A construção de software é um *criativo* processo. A metodologia do som pode fortalecer e liberar a mente criativa; não pode inflamar ou inspirar o trabalho pesado.

As diferenças não são menores - é como Salieri e Mozart. Estudo após estudo mostra que os melhores designers produzem estruturas que são mais rápidas, menores, mais simples, mais limpas e produzidas com menos esforço. As diferenças entre o ótimo e o médio se aproximam de uma ordem de magnitude.

Uma pequena retrospecção mostra que embora muitos sistemas de software úteis tenham sido projetados por comitês e construídos por projetos de várias partes, aqueles sistemas de software que têm entusiasmado fãs apaixonados são aqueles que são produtos de uma ou algumas mentes projetistas, grandes designers. Considere Unix, APL, Pascal, Modula, a interface Smalltalk, evenFortran; e contraste com Cobol, PL / I, Algol, MVS / 370 e MS-DOS (Fig. 16.1).

Portanto, embora eu apoie fortemente os esforços de transferência de tecnologia e desenvolvimento de currículo em andamento, acho que o esforço mais importante que podemos realizar é desenvolver maneiras de desenvolver grandes designers.

Nenhuma organização de software pode ignorar esse desafio. Bons gerentes, por mais raros que sejam, não são mais raros do que bons

E isso é	Não
Unix	Cobol
APL	PL / 1
Pascal	Algol
Módulo	MVS / 370
Conversa fiada	MS-DOS
Fortran	

Fig. 16.1 Produtos emocionantes

signatários. Grandes designers e grandes gerentes são muito raros. A maioria das organizações despende esforços consideráveis para encontrar e cultivar as perspectivas de gerenciamento; Não conheço ninguém que gaste igual esforço em encontrar e desenvolver os grandes designers dos quais a excelência técnica dos produtos dependerá em última instância.

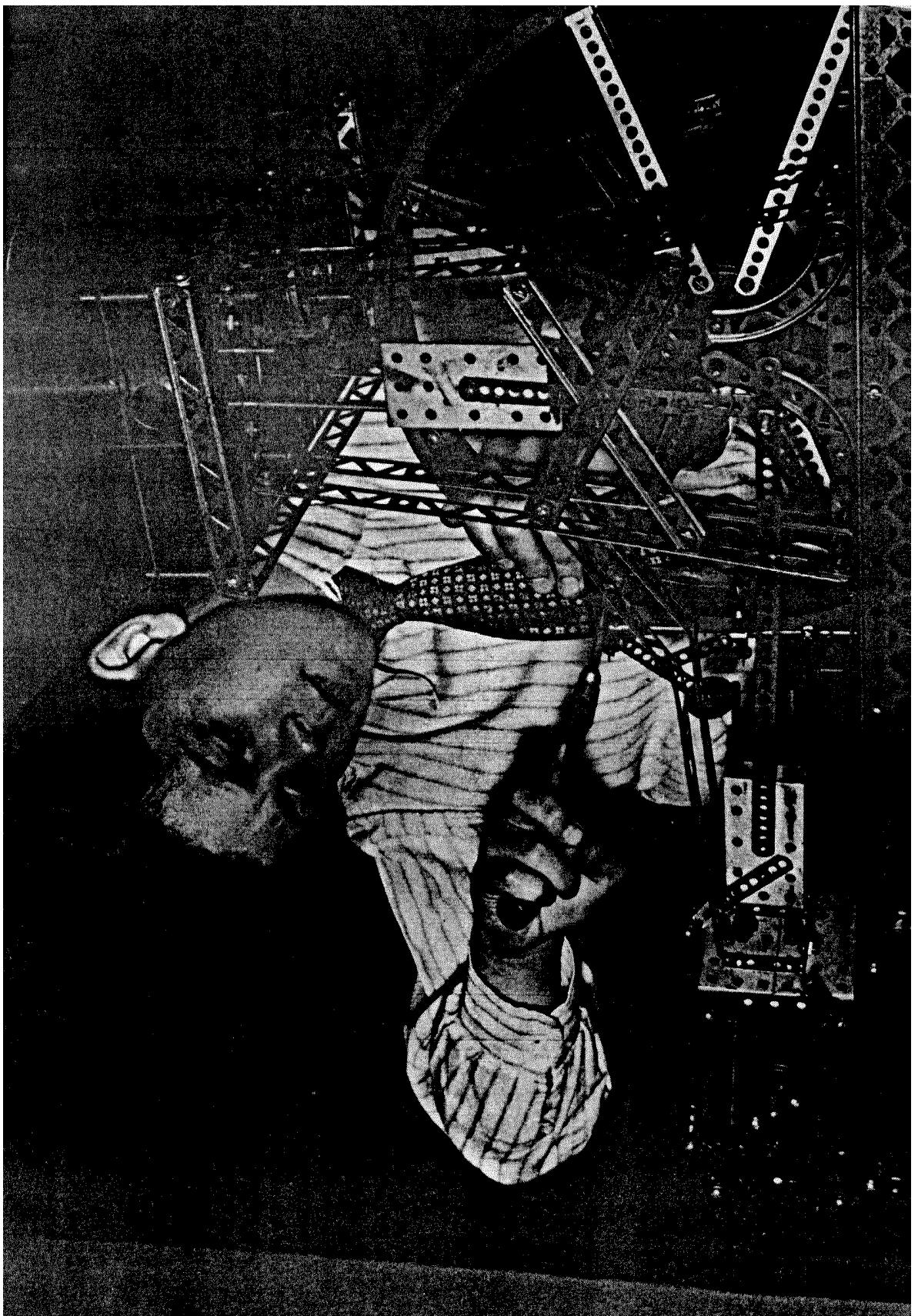
Minha primeira proposta é que cada organização de software deve determinar e proclamar que grandes designers são tão importantes para seu sucesso quanto os grandes gerentes, e que se pode esperar que eles sejam alimentados e recompensados da mesma forma. Não apenas o salário, mas as vantagens de reconhecimento - tamanho do escritório, mobília, equipamento técnico pessoal, fundos para viagens, apoio de pessoal - devem ser totalmente equivalentes.

Como desenvolver grandes designers? O espaço não permite uma discussão longa, mas algumas etapas são óbvias:

- Identifique sistematicamente os principais designers o mais cedo possível. Os melhores geralmente não são os mais experientes.
- Designe um mentor de carreira para ser responsável pelo desenvolvimento do cliente potencial e mantenha um arquivo de carreira cuidadoso.
- Elabore e mantenha um plano de desenvolvimento de carreira para cada cliente em potencial, incluindo estágios cuidadosamente selecionados com os melhores designers, episódios de educação formal avançada e cursos de curta duração, todos intercalados com design individual e atribuições de liderança técnica.
- Fornece oportunidades para designers em crescimento interagirem e estimularem uns aos outros.

17

"Não Silver Bullet "Refire



17

"Não Silver Bullet "Refired

Cada bala tem seu tarugo.

WILLIAM III DA INGLATERRA, PRÍNCIPE DE LARANJA

*Quem pensa uma peça impecável para ver,
Pensa o que nunca foi, nem é, nem será.*

ALEXANDER POPE, UM ENSAIO SOBRE CRÍTICA

Montando uma estrutura de peças prontas, 1945
The Bettman Archive

Sobre lobisomens e outros terroristas lendários

"No Silver Bullet - Essence and Accidents of Software Engineering" (agora Capítulo 16) foi originalmente um artigo convidado para o IFIP '86 conferência em Dublin, e foi publicado nesses anais.¹ *Computador* a revista o reimprimiu, atrás de uma capa gótica, ilustrada com fotos de filmes como *O Lobisomem de Londres*.² Eles também forneceram uma barra lateral explicativa "Para Matar o Lobisomem", apresentando a lenda (moderna) de que apenas balas de prata servem. Eu não estava ciente da barra lateral e das ilustrações antes da publicação, e nunca esperei que um artigo técnico sério fosse tão embelezado.

Computadores os editores eram especialistas em alcançar o efeito desejado, no entanto, e muitas pessoas parecem ter lido o artigo. Portanto, escolhi outra imagem de lobisomem para esse capítulo, uma antiga representação de uma criatura quase cômica. Espero que esta imagem menos berrante tenha o mesmo efeito salutar.

Existe também uma bala de prata - E AQUI ESTÁ!

"No Silver Bullet" afirma e argumenta que nenhum desenvolvimento de engenharia de software irá produzir uma melhoria de ordem de magnitude na produtividade da programação dentro de dez anos (a partir da publicação do artigo em 1986). Estamos agora nove anos nessa década, então é hora de ver como essa previsão está se mantendo.

Enquanto que *The Mythical Man-Month* gerou manicitações, mas poucos argumentos, "No Silver Bullet" ocasionou refutação de artigos, cartas aos editores de periódicos e cartas e ensaios que continuam até hoje.³ A maioria deles ataca o argumento central de que não há solução mágica e minha opinião clara de que não pode haver uma. A maioria concorda com a maioria dos argumentos em "NSB", mas depois afirma que existe de fato uma bala de prata para a besta do software, que o autor inventou. Ao reler as primeiras respostas de hoje, não posso deixar de notar que as panacéias empurradas com tanto vigor em 1986 e 1987 não tiveram os efeitos dramáticos alegados.

Eu compro hardware e software principalmente pelo teste do "usuário feliz" - conversas com *genuíno* clientes que pagam à vista que usam o produto e estão satisfeitos. Da mesma forma, devo acreditar mais facilmente que uma bala de prata se materializou quando um *genuíno* usuário independente avança e diz: "eu usei essa metodologia, ferramenta ou produto, e isso me deu uma melhoria dez vezes maior na produtividade do software. "

Muitos correspondentes fizeram emendas ou esclarecimentos válidos. Alguns empreenderam uma análise e refutação ponto a ponto, pelo que sou grato. Neste capítulo, compartilharei as melhorias e tratarei das refutações.

Escrita obscura será mal compreendida

Alguns escritores mostram que não consegui esclarecer alguns argumentos.

Acidente. O argumento central de "NSB" é tão claramente declarado no Resumo do Capítulo 16 quanto sei como colocá-lo. Alguns ficaram confusos, no entanto, com os termos *acidente* e *accidental*, que seguem um uso antigo que remonta a Aristóteles.⁴ Por *accidental*, Eu não quis dizer *ocorrendo por acaso*, nem *infeliz*, mas mais cedo *incidental*, ou *appurtenant*.

Eu não denegriria as partes accidentais da construção de software; em vez disso, sigo a dramaturga inglesa, escritora de histórias de detetive e teóloga Dorothy Sayers ao ver toda a atividade criativa consistir em (1) a formulação dos construtos conceituais, (2) a implementação na mídia real e (3) a interatividade com os usuários reais você usa.⁵ A parte de construção de software que chamei *essência* é a elaboração mental da construção conceitual; a parte que chamei *acidente* é o seu processo de implementação.

Uma questão de fato. Parece-me (embora não para todos) que a veracidade do argumento central se resume a uma questão de fato: qual fração do esforço total do software está agora associada à representação precisa e ordenada da construção conceitual, e qual fração é a esforço de elaborar mentalmente os construtos? A descoberta e correção de falhas cai

parcialmente em cada fração, de acordo com se as falhas são conceituais, como não reconhecer alguma exceção, ou representacionais, como um erro de ponteiro ou um erro de alocação de memória.

É minha opinião, e isso é tudo, que a parte accidental ou representacional da obra caiu agora para cerca de metade ou menos do total. Visto que essa fração é uma questão de fato, seu valor poderia, em princípio, ser determinado por medição.⁶ Caso contrário, minha estimativa pode ser corrigida por estimativas mais bem informadas e mais atuais. Significativamente, ninguém que escreveu publicamente ou em particular afirmou que a parte accidental é tão grande quanto 9/10.

O "NSB" argumenta, indiscutivelmente, que se a parte accidental da obra for inferior a 9/10 do total, encolhendo-a a zero (o que *seria* (veja a mágica) não dará uma ordem de magnitude de melhoria de produtividade. 1 *deve* atacar a essência.

Desde "NSB", Bruce Blum chamou minha atenção para o trabalho de Herzberg, Mausner e Sayderman em 1959.⁷ Eles descobrem que os fatores motivacionais *cão* aumentar a produtividade. Por outro lado, os fatores ambientais e accidentais, por mais positivos que sejam, não podem; mas esses fatores podem diminuir a produtividade quando negativos. "NSB" argumenta que grande parte do progresso do software foi a remoção de tais fatores negativos: linguagens de máquina incrivelmente estranhas, processamento em lote com longos tempos de resposta, ferramentas ruins e severas restrições de memória.

São as *essenciais*/dificuldades portanto *sem esperança*? Um excelente artigo de Brad Cox, de 1990, "Há uma bala de prata", argumenta eloquentemente em favor da abordagem de componentes reutilizáveis e intercambiáveis como um ataque à essência conceitual do problema.⁸ Concordo com entusiasmo.

Cox, entretanto, interpreta mal "NSB" em dois pontos. Primeiro, ele lê como uma afirmação de que as dificuldades de software surgem "de alguma deficiência em como os programadores criam software hoje". Meu argumento era que as dificuldades essenciais são inerentes à complexidade conceitual das funções do software a serem projetadas e construídas a qualquer momento, por qualquer método. Em segundo lugar, ele (e

outros) lêem "NSB" como afirmando que não há esperança de atacar as dificuldades essenciais da construção de software. Não era a minha intenção. Elaborar a construção conceitual de fato tem como dificuldades inerentes a complexidade, a conformidade, a mutabilidade e a invisibilidade. Os problemas causados por cada uma dessas dificuldades podem, no entanto, ser amenizados.

A complexidade ocorre por níveis. Por exemplo, a complexidade é a dificuldade inerente mais séria, mas nem toda complexidade é inevitável. Muito, mas não toda, a complexidade conceitual em nossas construções de software vem da complexidade arbitrária dos próprios aplicativos. Na verdade, Lars S0dahl da MYSIGMA S0dahl and Partners, uma empresa multinacional de consultoria de gestão, escreve:

Em minha experiência, a maioria das complexidades encontradas no trabalho de sistemas são sintomas de mau funcionamento organizacional. Tentar modelar essa realidade com programas igualmente complexos é, na verdade, conservar a bagunça em vez de resolver os problemas.

Steve Lukasik, da Northrop, argumenta que mesmo a complexidade organizacional talvez não seja arbitrária, mas pode ser suscetível a princípios de ordenação:

I treinado como físico e, portanto, ver as coisas "complexas" como suscetíveis de descrição em termos de conceitos mais simples. Agora você pode estar certo; Não vou afirmar que todas as coisas complexas são suscetíveis a princípios ordenadores. . . pelas mesmas regras de argumento, você não pode afirmar que eles podem não ser.

. . . A complexidade de ontem é a ordem de amanhã. A complexidade de a desordem molecular deu lugar à teoria cinética dos gases e às três leis da termodinâmica. Agora, o software pode nunca revelar esses tipos de princípios de ordenação, mas cabe a você explicar o porquê. Não estou sendo obtuso ou argumentativo. Acredito que algum dia a "complexidade" do software será entendida em termos de algumas noções de ordem superior (invariantes para o físico).

Não empreendi a análise mais profunda que Lukasik pede de maneira adequada. Como disciplina, precisamos de uma teoria da informação estendida que quantifique o conteúdo da informação de estruturas estáticas, assim como a teoria de Shannon faz para fluxos comunicados. Isso está muito além de mim. A Lukasik eu simplesmente respondo que a complexidade do sistema é uma função de uma miríade de detalhes que devem ser especificados exatamente, seja por alguma regra geral ou detalhe por detalhe, mas não apenas estatisticamente. Parece muito improvável que obras descoordenadas de muitas mentes tenham coerência suficiente para serem exatamente descritas por regras gerais.

Muito da complexidade em uma construção de software, entretanto, não se deve à conformidade com o mundo externo, mas sim à própria implementação - suas estruturas de dados, seus algoritmos, sua conectividade. O desenvolvimento de software em blocos de nível superior, criado por outra pessoa ou reutilizado do próprio passado, evita enfrentar camadas inteiras de complexidade. "NSB" defende um ataque sincero ao problema da complexidade, bastante otimista de que o progresso pode ser feito. Ele defende adicionar a complexidade necessária a um sistema de software:

- Hierarquicamente, por módulos ou objetos em camadas
- De forma incremental, para que o sistema funcione sempre.

Análise de Harel

David Harel, no jornal de 1992, "Biting the Silver Bullet", empreende a análise mais cuidadosa de "NSB" que foi publicada.⁹

Pessimismo vs. otimismo vs. realismo. Harel vê tanto "NSB" quanto "Software Aspects of Strategic Defense Systems", de Parnas, de 1984,¹⁰ como "muito sombrio". Portanto, ele pretende iluminar o lado mais brilhante da moeda, com o subtítulo de seu artigo "Rumo a um futuro mais brilhante para o desenvolvimento de sistemas". Tanto Cox quanto Harel lêem "NSB" como pessimista, e ele diz: "Mas se você ver esses mesmos fatos de uma nova perspectiva, uma conclusão mais otimista emerge." Ambos interpretaram mal o tom.

Em primeiro lugar, minha esposa, meus colegas e meus editores descobrem que eu erro com muito mais frequência no otimismo do que no pessimismo. Afinal, sou um programador por formação, e o otimismo é uma doença ocupacional de nosso ofício.

"NSB" diz explicitamente "Ao olharmos para o horizonte daqui a uma década, não vemos nenhuma bala de prata ... Ceticismo não é pessimismo, entretanto ... Não há uma estrada real, mas há uma estrada." Prevê que as inovações em curso em 1986, se desenvolvidas e exploradas, iriam *juntos* de fato, alcançar uma melhoria da ordem de magnitude na produtividade. À medida que a década de 1986-1996 avança, essa previsão parece, no mínimo, muito otimista, em vez de muito sombria.

Mesmo se "NSB" fosse universalmente visto como pessimista, o que há de errado nisso? A afirmação de Einstein de que nada pode viajar mais rápido do que a velocidade da luz é "sombria" ou "sombria"? Que tal o resultado de Gödel de que algumas coisas não podem ser calculadas? O "NSB" compromete-se a estabelecer que "a própria natureza do software torna improvável que haja balas de prata". Turski, em seu excelente artigo de resposta na Conferência IFIP, disse eloquentemente:

De todos os esforços científicos equivocados, nenhum é mais patético do que a busca pela pedra filosofal, uma substância que supostamente transforma metais básicos em ouro. O objetivo supremo da alquimia, perseguido ardenteamente por gerações de pesquisadores generosamente financiados por governantes seculares e espirituais, é um extrato puro de pensamento positivo, da suposição comum de que as coisas são como gostaríamos que fossem. É uma crença muito humana. É preciso muito esforço para aceitar a existência de problemas insolúveis. A vontade de ver uma saída, contra todas as probabilidades, mesmo quando se prove que ela não existe, é muito, muito forte. E a maioria de nós tem muita simpatia por essas almas corajosas que tentam alcançar o impossível. E assim continua. Estão sendo escritas dissertações sobre a quadratura de um círculo. Loções para restaurar o cabelo perdido são preparadas e vendem bem.

*Freqüentemente, estamos inclinados a seguir nosso próprio otimismo (ou explorar as esperanças otimistas de nossos patrocinadores). Com demasiada frequência, estamos dispostos a ignorar a voz da razão e dar ouvidos aos gritos de sereia dos traficantes de panaceia.*ⁿ

Turski e eu insistimos que sonhar com cachimbo *inibe o progresso e desperdiça esforços*.

Temas "Gloom". Harel percebe que a melancolia em "NSB" surge de três temas:

- Separação nítida em essência e acidente
- Tratamento de cada candidato bala de prata isoladamente
- Prever por apenas 10 anos, em vez de um tempo longo o suficiente para "esperar qualquer melhoria significativa".

Quanto ao primeiro, esse é o ponto principal do artigo. Ainda acredito que essa separação é absolutamente central para entender por que o software é difícil. É um guia seguro sobre os tipos de ataques a serem feitos.

Quanto ao tratamento de balas candidatas de forma isolada, "NSB" o faz de fato. Os vários candidatos foram propostos um a um, com reivindicações extravagantes para cada um *por si próprio*. É justo avaliá-los um por um. Não são as técnicas que me oponho, é esperar que façam mágica. Glass, Vessey e Conger em seu artigo de 1992 oferecem ampla evidência de que a vã busca por uma bala de prata ainda não terminou.¹²

Quanto à escolha de 10 anos contra 40 anos como um período de previsão, o período mais curto foi em parte uma concessão de que nossos poderes de previsão nunca foram bons além de uma década. Qual de nós em 1975 previu a revolução do microcomputador na década de 1980?

Existem outras razões para o limite de década: as alegações feitas para as balas candidatas todas tiveram um certo imediatismo sobre elas. Não me lembro de nada que dissesse "Invista na minha panacéia e você começará a ganhar depois de 10 anos". Além disso, as relações desempenho / preço de hardware melhoraram talvez cem vezes por década, e a comparação, embora bastante inválida, é

subconscientemente inevitável. Certamente faremos um progresso substancial nos próximos 40 anos; uma ordem de magnitude ao longo de 40 anos dificilmente é mágica.

O experimento mental de Harel. Harel propõe um experimento de pensamento em que postula "NSB" como tendo sido escrito em 1952, em vez de 1986, mas afirmando as mesmas proposições. Isso ele usa como um *reduto ad absurdum* argumentar contra a tentativa de separar a essência do acidente.

O argumento não funciona. Em primeiro lugar, "NSB" começa afirmando que as dificuldades accidentais dominaram grosseiramente as essenciais na programação dos anos 1950, que não o fazem mais e que eliminá-las resultou em melhorias de ordem de magnitude. Traduzir esse argumento 40 anos atrás não é razoável; dificilmente se pode imaginar afirmar em 1952 que as dificuldades accidentais não ocasionam grande parte do esforço.

Em segundo lugar, o estado de coisas que Harel imagina ter prevalecido na década de 1950 é impreciso:

Essa foi a época em que, em vez de lutar com o design de sistemas grandes e complexos, os programadores estavam no negócio de desenvolver programas convencionais de uma pessoa (que seriam da ordem de 100-200 linhas em uma linguagem de programação moderna) que deveriam realizar tarefas algorítmicas limitadas. Dada a tecnologia e metodologia disponíveis na época, tais tarefas eram igualmente formidáveis. Falhas, erros e prazos perdidos estavam por toda parte.

Ele então descreve como as falhas postuladas, erros e prazos perdidos nos pequenos programas convencionais de uma pessoa foram melhorados em uma ordem de magnitude ao longo dos próximos 25 anos.

Mas o estado da arte na década de 1950 não eram, na verdade, pequenos programas para uma pessoa. Em 1952, a Univac estava trabalhando no processamento do censo de 1950 com um programa complexo desenvolvido por cerca de oito programadores.¹³ Outras máquinas estavam fazendo dinâmica química, cálculos de difusão de nêutrons, cálculos de desempenho de mísseis, etc.¹⁴ Montadores, realocando linkers e carregadores, sistemas interpretativos de ponto flutuante, etc. estavam em uso rotineiro.^{quinze}

Em 1955, as pessoas estavam criando programas de negócios de 50 a 100 homens por ano.¹⁶ Em 1956, a General Electric tinha em operação um sistema de folha de pagamento em sua fábrica de eletrodomésticos em Louisville com mais de 80.000 palavras de programa. Em 1957, o computador de defesa aérea SAGE ANFSQ / 7 estava funcionando há dois anos, e um sistema de tempo real duplex protegido contra falhas e baseado em comunicação de 75.000 instruções estava em operação em 30 locais.¹⁷ Dificilmente se pode sustentar que é a evolução das técnicas para programas de uma pessoa que descreve principalmente os esforços de engenharia de software desde 1952.

E AQUI ESTÁ. Harel passa a oferecer sua própria bala de prata, uma técnica de modelagem chamada "The Vanilla Framework". A abordagem em si não é descrita em detalhes suficientes para avaliação, mas é feita referência a um artigo e a um relatório técnico que aparecerá em livro no devido tempo.¹⁸ A modelagem aborda a essência, a elaboração e depuração adequadas de conceitos, portanto, é possível que o framework Vanilla seja revolucionário. Espero que sim. Ken Brooks relata que descobri que é uma metodologia útil quando a experimentei para uma tarefa real.

Invisibilidade. Harel argumenta fortemente que muito da construção conceitual de software é inherentemente topológica por natureza e essas relações têm contrapartes naturais em representações espaciais / gráficas:

O uso de formalismos visuais apropriados pode ter um efeito espetacular em engenheiros e programadores. Além disso, esse efeito não se limita a meros problemas acidentais; a qualidade e rapidez de seus próprios pensamento foi considerado melhorado. O desenvolvimento de sistema bem-sucedido no futuro girará em torno de representações visuais. Vamos primeiro conceituar, usando as entidades e relações "adequadas", e então formular e reformular nossas concepções como uma série de modelos cada vez mais abrangentes representados em uma combinação apropriada de linguagens visuais. Deve ser uma combinação, uma vez que os modelos de sistema têm várias facetas, cada uma das quais evoca diferentes tipos de imagens mentais.

.... *Alguns aspectos do processo de modelagem não foram vindo como os outros, prestando-se a uma boa visualização. Operações algorítmicas em variáveis e estruturas de dados, por exemplo, provavelmente permanecerão textuais.*

Harel e eu somos muito próximos. O que argumentei é que a estrutura do software não está embutida no espaço tridimensional, portanto, não há um único mapeamento natural de um projeto conceitual para um diagrama, seja em duas dimensões ou mais. Ele admite, e eu concordo, que são necessários vários diagramas, cada um cobrindo algum aspecto distinto, e que alguns aspectos não são bem diagramas.

Eu compartilho completamente seu entusiasmo por usar diagramas como recursos de reflexão e design. Há muito gosto de perguntar a programadores candidatos: "Onde é o próximo novembro?" Se a pergunta for muito enigmática, então, "Fale-me sobre seu modelo mental do calendário." Os programadores realmente bons têm sentidos espaciais fortes; eles geralmente têm modelos geométricos de tempo; e muitas vezes entendem a primeira pergunta sem elaboração. Eles têm modelos altamente individualistas.

Jones's Point - Produtividade Segue Qualidade

Capers Jones, escrevendo primeiro em uma série de memorandos e depois em um livro, oferece uma visão penetrante, que foi declarada por vários de meus correspondentes. "NSB", como a maioria dos escritos da época, era focado em *produtividade*, a saída do software por unidade de entrada. Jones diz: "Não. Concentre-se em *qualidade*, e a produtividade virá em seguida. "¹⁹ Ele argumenta que projetos caros e atrasados investem a maior parte do trabalho e tempo extras para encontrar e reparar erros na especificação, no projeto, na implementação. Ele oferece dados que mostram uma forte correlação entre a falta de controles sistemáticos de qualidade e desastres de cronograma. Eu acredito nisso. Boehm aponta que a produtividade cai novamente à medida que se busca qualidade extrema, como no software do ônibus espacial da IBM.

Coqui também argumenta que disciplinas sistemáticas de desenvolvimento de software foram desenvolvidas em resposta a questões de qualidade

(especialmente a prevenção de grandes desastres) em vez de preocupações com a produtividade.

Mas observe: o objetivo de aplicar os princípios da Engenharia à produção de software na década de 1970 era aumentar a qualidade, a testabilidade, a estabilidade e a previsibilidade dos produtos de software. - não necessariamente a eficiência da produção de software.

A força motriz para usar os princípios da Engenharia de Software na produção de software foi o medo de acidentes graves que poderiam ser causados por artistas incontroláveis responsáveis pelo desenvolvimento de sistemas cada vez mais complexos. vinte

Então, o que aconteceu com a produtividade?

Números de produtividade. Os números de produtividade são muito difíceis de definir, calibrar e encontrar. Capers Jones acredita que para dois programas COBOL equivalentes escritos com 10 anos de diferença, um sem metodologia estruturada e outro com, o ganho é para fator de três.

Ed Yourdon diz: "Vejo as pessoas obtendo uma melhoria cinco vezes maior devido às estações de trabalho e ferramentas de software." Tom DeMarco acredita que "Sua expectativa de uma melhoria da ordem de magnitude em 10 anos, devido a toda a cesta de técnicas, era otimista. Não vi organizações fazendo uma melhoria da ordem de magnitude."

Software encolhido - Compre; não construa. Uma avaliação de 1986 no "NSB" provou, eu acho, estar correta: "O desenvolvimento do mercado de massa é ... A tendência de longo prazo mais profunda na engenharia de software." Do ponto de vista da disciplina, o software para o mercado de massa é quase um novo setor em comparação com o de desenvolvimento de software customizado, seja interno ou externo. Quando os pacotes vendem aos milhões - ou mesmo aos milhares - qualidade, pontualidade, desempenho do produto e custo de suporte se tornam questões dominantes, ao invés do custo de desenvolvimento que é tão crucial para o cliente systems.

Ferramentas poderosas para a mente. A maneira mais dramática de melhorar a produtividade dos programadores de sistemas de informações de gerenciamento (MIS) é ir até a loja de informática local e comprar na prateleira o que eles teriam construído. Isso não é ridículo; a disponibilidade de software barato e poderoso encolhido atendeu a muitas necessidades que antes gerariam pacotes personalizados. Essas ferramentas elétricas para a mente são mais como furadeiras elétricas, serras e lixadeiras do que grandes e complexas ferramentas de produção. A integração destes em conjuntos compatíveis e com ligações cruzadas, como o Microsoft Works e o ClarisWorks melhor integrado, oferece imensa flexibilidade. E como a coleção de ferramentas manuais elétricas do proprietário, o uso frequente de um pequeno conjunto, para muitas tarefas diferentes, desenvolve familiaridade. Essas ferramentas devem enfatizar a facilidade de uso para o usuário ocasional, não para o profissional.

Ivan Selin, presidente da American Management Systems, Inc., escreveu-me em 1987:

*Discordo com sua afirmação de que os pacotes não mudaram muito. . . :
Acho que você desconsiderou as principais implicações de sua observação de que, [os pacotes de software] "podem ser um pouco mais generalizados e um pouco mais personalizáveis do que antes, mas não muito." Mesmo aceitando essa afirmação pelo valor de face, acredito que os usuários veem os pacotes como sendo mais generalizados e mais fáceis de personalizar, e que essa percepção leva os usuários a serem muito mais receptivos aos pacotes. Na maioria dos casos que minha empresa descobre, são os usuários [finais], e não o pessoal do software, que relutam em usar pacotes porque pensam que perderão recursos ou funções essenciais e, portanto, a perspectiva de personalização fácil é um grande ponto de venda para eles.*

Acho que Selin está certo - subestimei o grau de personalização do pacote e sua importância.

Programação orientada a objetos - uma bala de metal serve?

Prédio com peças maiores. A ilustração que abre este capítulo nos lembra que, se alguém montar um conjunto de peças, cada uma

que podem ser complexos e todos projetados para ter interfaces suaves, estruturas bastante ricas se juntam rapidamente.

Uma visão da programação orientada a objetos é que é uma disciplina que impõe *modularidade* e interfaces limpas. Uma segunda visão enfatiza *encapsulamento*, o fato de não se poder ver, muito menos desenhar, a estrutura interna das peças. Outra visão enfatiza *herança*, com seu concomitante *hierárquico* estrutura de classes, com funções virtuais. Ainda outra visão enfatiza *forte digitação de dados abstratos*, com sua garantia de que um determinado tipo de dados será manipulado apenas por operações adequadas a ele.

Agora, qualquer uma dessas disciplinas pode ser adquirida sem levar o pacote Smalltalk ou C ++ inteiro - muitos deles são anteriores à tecnologia orientada a objetos. A atratividade da abordagem orientada a objetos é a de uma pílula multivitamínica: de uma só vez (ou seja, o retreinamento do programador), obtém-se todos eles. É um conceito muito promissor.

Por que a técnica orientada a objetos cresceu lentamente? Nas nove anos desde "NSB", a expectativa tem crescido constantemente. Por que o crescimento foi lento? As teorias abundam. James Coggins, autor por quatro anos da coluna, "The Best of comp.lang.c ++" em *O Relatório C +*, oferece esta explicação:

O problema é que os programadores em OO têm feito experiências em aplicações incestuosas e visando baixa abstração, em vez de alta. Por exemplo, eles vêm construindo classes como lista ligada ou definir em vez de classes como interface de usuário ou feixe de radiação ou modelo de elementos finitos. Infelizmente, a mesma forte verificação de tipo em C ++ que ajuda os programadores a evitar erros também torna difícil construir coisas grandes a partir de pequenos. vinte e um

Ele volta ao problema básico de software e argumenta que uma maneira de atender às necessidades de software não atendidas é aumentar o tamanho da força de trabalho inteligente, habilitando e cooptando nossos clientes. Thisarguesfort-downdesign: ...

“Nós projetamos classes de grande granularidade que abordam conceitos com os quais nossos clientes já estão trabalhando, eles podem entender e questionar o design conforme ele cresce e podem cooperar no design de casos de teste. Meus colaboradores da oftalmologia não se importam com as pilhas; eles se preocupam com as descrições da forma polinomial de Legendre das córneas. Pequenos encapsulamentos geram pequenos benefícios.

David Parnas, cujo artigo foi uma das origens dos conceitos orientados a objetos, vê a questão de forma diferente. Eu tenho me escrito:

A resposta é simples. É porque [OO] foi vinculado a uma variedade de linguagens complexas. Em vez de ensinar às pessoas que OO é um tipo de design e dar-lhes princípios de design, as pessoas ensinaram que OO é o uso de uma ferramenta específica. Podemos escrever programas bons ou ruins com qualquer ferramenta. A menos que ensinemos as pessoas a projetar, as linguagens importam muito pouco. O resultado é que as pessoas fazem projetos ruins com essas linguagens e obtêm muito pouco valor delas. Se o valor for pequeno, ele não pegará.

Custos antecipados, benefícios posteriores. Minha própria convicção é que as técnicas orientadas a objetos apresentam um caso peculiarmente grave de uma doença que caracteriza muitos aprimoramentos metodológicos. Os custos iniciais são muito substanciais - principalmente o retreinamento dos programadores para pensar de uma maneira totalmente nova, mas também o investimento extra em funções de modelagem em classes generalizadas. Os benefícios, que considero reais e não meramente putativos, ocorrem ao longo de todo o ciclo de desenvolvimento; mas os grandes benefícios compensam durante as atividades de construção, extensão e manutenção do sucessor. Coggins diz: "As técnicas orientadas a objetos não tornarão o desenvolvimento do primeiro projeto mais rápido, nem do próximo. O quinto dessa família será incrivelmente rápido." 22

Apostar dinheiro inicial real por causa dos benefícios projetados, mas duvidosos mais tarde, é o que os investidores fazem todos os dias. Em muitas organizações de programação, no entanto, requer verdadeira coragem gerencial, uma mercadoria muito mais escassa do que competência técnica ou proficiência administrativa. Acredito que o grau extremo de custo inicial e de retorno de benefício é o maior fator

retardando a adoção de técnicas OO. Mesmo assim, o C ++ parece estar constantemente substituindo o C em muitas comunidades.

E a reutilização?

A melhor maneira de atacar a essência da construção de software é não construí-lo de forma alguma. O software de pacote é apenas uma das maneiras de fazer isso. A reutilização de programas é outra. Na verdade, a promessa de fácil reutilização de classes, com fácil customização por herança, é um dos maiores atrativos das técnicas orientadas a objetos.

Como é frequentemente o caso, à medida que se obtém alguma experiência com *para* nova maneira de fazer negócios o novo modo não é tão simples quanto parece à primeira vista.

Claro, os programadores sempre reutilizaram seu próprio trabalho manual. Jones diz,

*A maioria dos programadores experientes possui bibliotecas privadas que lhes permitem desenvolver software com cerca de 30% de código reutilizado por volume. A reutilização no nível corporativo visa 75% código reutilizado por volume e requer biblioteca especial e suporte administrativo. O código reutilizável corporativo também implica mudanças nas práticas de contabilidade e medição do projeto para dar crédito à reutilização.*²³

W. Huang propôs organizar fábricas de software com uma gestão matricial de especialistas funcionais, de modo a aproveitar a propensão natural de cada um em reutilizar seu próprio código.²⁴

Van Snyder, do JPL, aponta para mim que a comunidade de software matemático tem uma longa tradição de reutilização de software:

Conjeturamos que as barreiras à reutilização não estão do lado do produtor, mas do lado do consumidor. Se um engenheiro de software, um consumidor potencial de componentes de software padronizados, perceber que é mais caro encontrar um componente que atenda às suas necessidades e, assim, verificar, do que escrever um novo, um novo componente duplicado será escrito. Observe que dissemos percebe acima. Não importa qual seja o verdadeiro custo da reconstrução.

A reutilização tem sido bem-sucedida para o software matemático por duas razões: (1) é misteriosa, exigindo uma enorme entrada intelectual por linha de código; e (2) há uma nomenclatura rica e padrão, ou seja, matemática, para descrever a funcionalidade de cada componente. Assim, o custo para reconstruir um componente de software matemático é alto e o custo para descobrir a funcionalidade de um componente existente é baixo. A longa tradição de periódicos profissionais publicando e coletando algoritmos, e oferecendo-os a um custo modesto, e interesses comerciais oferecendo algoritmos de qualidade muito alta a um custo um pouco mais alto, mas ainda modesto, torna a descoberta de um componente que atende às necessidades de alguém mais simples do que em muitas outras disciplinas, onde às vezes não é possível especificar uma necessidade precisa e concisa.

O mesmo fenômeno de reutilização é encontrado em várias comunidades, como aquelas que constroem códigos para reatores nucleares, modelos climáticos e modelos oceânicos, e pelos mesmos motivos. Cada uma das comunidades cresceu com os mesmos livros didáticos e notações padrão.

Como a reutilização em nível corporativo se sai hoje? Muito estudo; relativamente pouca prática nos Estados Unidos; relatos anedóticos de mais reutilização no exterior.²⁵

Jones relata que todos os clientes de sua empresa com mais de 5.000 programadores têm pesquisa formal de reutilização, enquanto menos de 10% dos clientes com menos de 500 programadores têm.²⁶ Ele relata que em setores com maior potencial de reutilização, a pesquisa de reutilização (não implantação) "é ativa e enérgica, mesmo que ainda não seja totalmente bem-sucedida". Ed Yourdon relata uma software house em Manila que tem 50 de seus 200 programadores construindo apenas módulos reutilizáveis para o restante usar; "Eu vi alguns casos - a adoção é devido a *organizacional*/fatores como a estrutura de recompensa, não fatores técnicos."

DeMarco me disse que a disponibilidade de pacotes para o mercado de massa e sua adequação como provedores de funções genéricas, como sistemas de banco de dados, reduziu substancialmente a pressão

e a utilidade marginal de reutilizar módulos do código de aplicação de alguém.
"Os módulos reutilizáveis tendem a ser as funções genéricas de qualquer maneira."

Parnas escreve,

Reutilizar é algo muito mais fácil de dizer do que fazer. Fazer isso requer um bom design e uma documentação muito boa. Mesmo quando vemos um bom design, o que ainda é raro, não veremos os componentes reutilizados sem uma boa documentação.

Ken Brooks comenta sobre a dificuldade de antecipar que a generalização se mostrará necessária: "Eu continuo tendo que dobrar as coisas mesmo no quinto uso de minha própria biblioteca de interface de usuário pessoal."

A verdadeira reutilização parece estar apenas começando. Jones relata que alguns módulos de código reutilizáveis estão sendo oferecidos no mercado aberto a preços entre 1% e 20% dos custos normais de desenvolvimento.²⁷ DeMarco diz,

Estou ficando muito desanimado com todo o fenômeno da reutilização. Há quase uma ausência total de um teorema de existência para reutilização. O tempo confirmou que há um grande gasto em tornar as coisas reutilizáveis.

Yourdon estima a grande despesa: "Uma boa regra prática é que esses componentes reutilizáveis demandarão o dobro do esforço de um componente 'único'."²⁸ Vejo essa despesa como exatamente o esforço de produzir o componente, discutido no Capítulo 1. Portanto, minha estimativa da relação de esforço seria três vezes maior.

Obviamente, estamos vendo muitas formas e variedades de reutilização, mas não tanto como esperávamos agora. Ainda há muito que aprender.

LearningLargeVocabularies - APredictable mas imprevisível problema para reutilização de software

Quanto mais alto o nível em que se pensa, mais numerosos são os elementos-pensamento primitivos com os quais devemos lidar. Tão profissional-

as linguagens de gramática são muito mais complexas do que as linguagens de máquina, e as linguagens naturais são ainda mais complexas. Linguagens de nível superior têm vocabulários maiores, sintaxe mais complexa e semântica mais rica.

Como disciplina, não ponderamos as implicações desse fato para a reutilização de programas. Para melhorar a qualidade e a produtividade, queremos construir programas compondo pedaços de função depurada que são substancialmente maiores do que as instruções em linguagens de programação. Portanto, quer façamos isso por bibliotecas de classes de objetos ou bibliotecas de procedimentos, devemos encarar o fato de que estamos aumentando radicalmente o tamanho de nossos vocabulários de programação. A aprendizagem do vocabulário constitui uma grande parte da barreira intelectual a ser reutilizada.

Então, hoje as pessoas têm bibliotecas de classe com mais de 3.000 membros. Muitos objetos requerem especificação de 10 a 20 parâmetros e variáveis de opção. Qualquer pessoa que estiver programando com essa biblioteca deve aprender a sintaxe (as interfaces externas) e a semântica (o comportamento funcional detalhado) de seus membros se quiserem atingir todo o potencial de reutilização.

Essa tarefa está longe de ser inútil. Os falantes nativos usam rotineiramente vocabulários de mais de 10.000 palavras, pessoas instruídas muito mais. De alguma forma, aprendemos a sintaxe e a semântica muito sutil. Nós diferenciamos corretamente entre *gigante*, *enorme*, *vasto*, *enorme*, *mamute*; as pessoas simplesmente não falam de desertos gigantescos ou enormes elefantes.

Precisamos de pesquisa para se apropriar para o problema de reutilização de software o grande corpo de conhecimento sobre como as pessoas adquirem a linguagem. Algumas das lições são imediatamente óbvias:

- As pessoas aprendem em contextos de frase, por isso precisamos publicar muitos exemplos de produtos compostos, não apenas bibliotecas de partes.
- As pessoas não memorizam nada além da ortografia. Eles aprendem sintaxe e semântica de forma incremental, no contexto, pelo uso. As pessoas agrupam
- regras de composição de palavras por classes sintáticas, não por subconjuntos compatíveis de objetos.

Net on Bullets - Posição inalterada

Portanto, voltamos aos fundamentos. Complexidade é o negócio em que atuamos, e a complexidade é o que nos limita. RL Glass, escrevendo em 1988, resume com precisão minhas visões de 1995:

SW o que, em retrospecto, Parnas e Brooks nos disseram? O desenvolvimento de software é um negócio conceitualmente difícil. Que as soluções mágicas não estão ao virar da esquina. Que é hora de o praticante examinar as melhorias evolutivas ao invés de esperar - ou esperança - para os revolucionários.

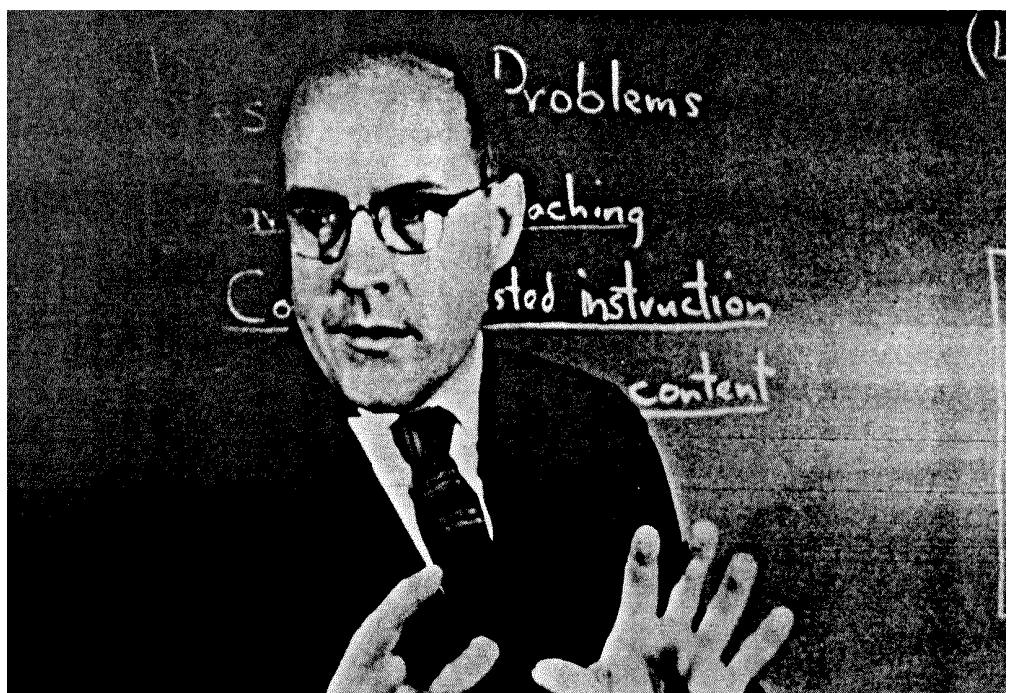
Alguns na área de software acham que esta é uma imagem desanimadora. Eles são os que ainda pensavam que os avanços estavam próximos.

Mas alguns de nós - aqueles de nós duros o suficiente para pensar que somos realistas - veja isso como uma lufada de ar fresco. Por fim, podemos nos concentrar em algo um pouco mais viável do que uma torta no céu. Agora, talvez, possamos continuar com as melhorias incrementais possíveis na produtividade do software, em vez de esperar pelos avanços que provavelmente nunca acontecerão.²⁹

18

*Proposições de o
Homem-mês mítico:*

Verdadeiro ou falso?



18

Proposições de o Mítico Homem-Mês Verdadeiro ou falso?

*Para resumir é muito bom,
Onde estamos ou não somos compreendidos.*

SAMUEL BUTLER. Hudibras

**Brooks afirmando uma proposição, 1967 Foto
de J. Alex Langley para *Fortuna Revista***

Sabe-se muito mais hoje sobre engenharia de software do que em 1975. Quais das afirmações na edição original de 1975 foram apoiadas por dados e experiência? Quais foram refutadas? O que ficou obsoleto com as mudanças do mundo? (Você pode perguntar: "Se isso é tudo que o livro original disse, por que demorou 177 páginas para dizê-lo?") Os comentários entre colchetes são novos.

A maioria dessas proposições é operacionalmente testável. Minha esperança ao apresentá-los de forma nítida é enfocar os pensamentos, medições e comentários dos leitores.

Capítulo 1. O poço de alcatrão

1,1 Um produto de sistema de programação exige cerca de nove vezes mais esforço do que os programas componentes escritos separadamente para uso privado. Estimo que a produção impõe um fator de três; e que projetar, integrar e testar componentes em um sistema coerente impõe um fator de três; e que esses componentes de custo são essencialmente independentes uns dos outros.

1,2 A arte da programação "gratifica anseios criativos construídos profundamente dentro de nós e encanta as sensibilidades que temos em comum com todos os homens", proporcionando cinco tipos de alegrias:

- A alegria de fazer coisas
- A alegria de fazer coisas que são úteis para outras pessoas

• O fascínio de criar objetos parecidos com quebra-cabeças de peças móveis interligadas

- A alegria de sempre aprender, de uma tarefa que não se repete
- «O deleite de trabalhar num meio tão dócil - puro pensamento - que, no entanto, existe, se move e funciona de uma forma que os objetos-palavra não o fazem.

1.3 Da mesma forma, a embarcação tem problemas especiais inerentes a ela.

- Ajustar-se à exigência de perfeição é a parte mais difícil de aprender a programar.

"Outros definem seus objetivos e devemos depender de coisas (especialmente programas) que não podemos controlar; a autoridade não é igual à responsabilidade.

«Isso soa pior do que é: a autoridade real vem do momento de realização.

 - Com qualquer criatividade, venha horas monótonas de trabalho árduo; a programação não é exceção.
 - * O projeto de programação converge mais lentamente quanto mais perto se chega do fim, ao passo que se espera que convirja mais rápido à medida que se aproxima do fim.
 - * O produto de alguém está sempre ameaçado de obsolescência antes de ser concluído. O tigre real nunca é páreo para o de papel, a menos que o uso real seja desejado.

Capítulo 2. O Mítico Homem-Mês

- 2,1 Mais projetos de programação deram errado por falta de tempo no calendário do que por todas as outras causas combinadas.
- 2,2 Uma boa cozinha leva tempo; algumas tarefas não podem ser apressadas sem prejudicar o resultado.
- 2,3 Todos os programadores são otimistas: "Tudo irá bem." Como o
- 2,4 programador constrói com puro pensamento, esperamos poucas dificuldades na implementação.
- 2,5 Mas o nosso *Ideias* próprios estão com defeito, então temos bugs.
- 2,6 Nossas técnicas de estimativa, construídas em torno da contabilidade de custos, confundem esforço e progresso. *O homem-mês é um mito falacioso e perigoso, pois implica que homens e meses são intercambiáveis.*
- 2,7 Dividir uma tarefa entre várias pessoas ocasiona um esforço extra de comunicação - treinamento e intercomunicação.
- 2,8 Minha regra é 1/3 do cronograma para design, 1/6 para codificação, 1/4 para teste de componentes e 1/4 para teste de sistema.

- 2,9 Como disciplina, não temos dados de estimativa.
- 2,10 Como não temos certeza sobre nossas estimativas de programação, muitas vezes não temos coragem de defendê-las obstinadamente contra a pressão da gerência e do cliente.
- 2,11 Lei de Brooks: Adicionar mão de obra a um projeto de software atrasado o torna mais tarde.
- 2,12 Adicionar pessoas a um projeto de software aumenta o esforço total necessário de três maneiras: o trabalho e a interrupção do reparticionamento, o treinamento de novas pessoas e a intercomunicação agregada.

Capítulo 3. A Equipe Cirúrgica

- 3,1 Muito bons programadores profissionais são *tem tempos* tão produtivos quanto os pobres, no mesmo nível de treinamento e experiência de dois anos. (Sackman, Grant e Erickson) Os dados de Sackman, Grant e Erickson não mostraram qualquer correlação entre experiência e desempenho. Duvido da universalidade desse resultado.
- 3,2
- 3,3 Uma equipe pequena e afiada é melhor - com o mínimo de mentes possível. Uma
- 3,4 equipe de duas pessoas, com um líder, costuma ser o melhor uso das mentes.
[Observe o plano de Deus para o casamento.]
- 3,5 Uma equipe pequena e afiada é muito lenta para sistemas realmente
- 3,6 grandes. A maioria das experiências com sistemas realmente grandes mostra a abordagem de força bruta para aumentar a escala para ser cara, lenta, ineficiente e para produzir sistemas que não são conceitualmente integrados.
- 3,7 Um programador-chefe, organização de equipe cirúrgica oferece uma maneira de obter a integridade do produto de poucas mentes e a produtividade total de muitos ajudantes, com comunicação radicalmente reduzida.

Capítulo 4. Aristocracia, Democracia e Design de Sistema

- 4,1 "Integridade conceitual é a consideração mais importante no design do sistema. "

- 4,2 "O *Razão* da função à complexidade conceitual é o teste final eu do projeto do sistema ", não apenas a riqueza da função. [Esta proporção é uma medida de facilidade de uso, válida tanto para usos simples quanto difíceis.]
- 4,3 Para atingir a integridade conceitual, um projeto deve partir de uma mente ou de um pequeno grupo de mentes concordantes.
- 4,4 "Separar o esforço arquitetônico da implementação é uma maneira muito poderosa de obter integração conceitual em projetos muito grandes." [Pequenos também.]
- 4,5 "Se um sistema deve ter integridade conceitual, alguém deve controlar os conceitos. Essa é uma aristocracia que não precisa de desculpas."
- 4,6 A disciplina é boa para a arte. A provisão externa de uma arquitetura aprimora, e não restringe, o estilo criativo de um grupo de implementação.
- 4,7 Um sistema conceitualmente integrado é mais rápido de construir e testar.
- 4,8 Grande parte da arquitetura, implementação e realização de software pode ocorrer em paralelo. [O design de hardware e software também pode prosseguir em paralelo.]

Capítulo 5. O efeito do segundo sistema

- 5,1 A comunicação inicial e contínua pode dar ao arquiteto boas leituras de custo e ao construtor confiança no projeto, sem obscurecer a divisão clara de responsabilidades.
- 5,2 Como um arquiteto pode influenciar com sucesso a implementação:
«Lembre-se que o construtor tem a responsabilidade criativa pela implementação; o arquiteto apenas sugere.
- Esteja sempre pronto para sugerir uma forma de implementar qualquer coisa que alguém especificar; esteja preparado para aceitar qualquer outra forma igualmente boa.
 - Trate essas sugestões de maneira discreta e privada.
- ⁹Esteja pronto para abrir mão de crédito por melhorias sugeridas.

- «Ouça as sugestões do construtor para melhorias na arquitetura.
- 5,3 O segundo é o sistema mais perigoso que uma pessoa já projeta; a tendência geral é superestabelecê-lo. OS / 360 é um
- 5,4 bom exemplo do efeito do segundo sistema. [Windows NT parece ser um exemplo dos anos 1990].
- 5,5 Atribuindo *a priori* valores em bytes e microssegundos para funções é uma disciplina que vale a pena.

Capítulo 6. Passando a Palavra

- 6,1 Mesmo quando a equipe de design é grande, os resultados devem ser reduzidos à escrita em um ou dois, para que as minidecisões sejam consistentes.
- 6,2 É importante definir explicitamente as partes de uma arquitetura que são *não* prescritos tão cuidadosamente quanto aqueles que
- 6,3 são. É necessária uma definição formal de design, para precisão, e uma definição em prosa para compreensão.
- 6,4 Uma das definições formais e em prosa deve ser padrão e a outra derivada. Qualquer definição pode servir em qualquer função.
- 6,5 Uma implementação, incluindo uma simulação, pode servir como uma definição arquitetônica; esse uso tem desvantagens formidáveis.
- 6,6 A incorporação direta é uma técnica muito limpa para impor um padrão de arquitetura em software. [Em hardware, também - considere o MacWIMPinterfacebuilt intoROM.] Uma
- 6,7 "definição arquitetônica será mais limpa e a disciplina [arquitetônica] mais rígida se pelo menos duas implementações forem criadas inicialmente."
- 6,8 É importante permitir interpretações telefônicas por um arquiteto em resposta às perguntas dos implementadores; é imperativo registrá-los e publicá-los. [O correio eletrônico agora é o meio de escolha.]
- 6,9 "O melhor amigo do gerente de projeto é seu adversário diário, a organização independente de teste de produtos."

Capítulo 7. Por que a Torre de Babel falhou?

7.1 O projeto da Torre de Babel falhou por falta de *comunicação* e de seu consequente, *organização*.

Comunicação

- 7,2 "Desastres de cronograma, desajustes funcionais e bugs de sistema surgem porque a mão esquerda não sabe o que a mão direita está fazendo." As equipes se distanciam em suposições. As equipes devem comunicar-se umas com as outras de todas as maneiras possíveis: informalmente, por meio de reuniões regulares do projeto com briefings técnicos e por meio de uma pasta de trabalho formal compartilhada do projeto. [E por correio eletrônico.]

Livro de Trabalho do Projeto

- 7,4 Uma pasta de trabalho do projeto "não é tanto um documento separado, mas uma estrutura imposta aos documentos que o projeto produzirá de qualquer maneira".
- 7,5 "*Tudo* os documentos do projeto precisam fazer parte dessa estrutura [da pasta de trabalho]."
- 7,6 A estrutura da pasta de trabalho precisa ser projetada *cuidadosamente* e *cedo*.
- 7,7 Estruturar adequadamente a documentação contínua desde o início "molda a escrita posterior em segmentos que se encaixam nessa estrutura" e melhorará os manuais do produto.
- 7,8 "*Cada* membro da equipe deve ver */á* o material [da pasta de trabalho]. "[Eu diria agora, cada membro da equipe *deve ser capaz* para ver tudo isso. Ou seja, as páginas da World-Wide Web seriam suficientes.]
- 7,9 A atualização oportuna é de importância crítica.
- 7,10 O usuário precisa ter atenção especial às mudanças desde sua última leitura, com observações sobre o seu significado.
- 7,11 A pasta de trabalho do Projeto OS / 360 começou com papel e mudou de tomicroficha.
- 7,12 Hoje [ainda em 1975], o caderno eletrônico compartilhado é um

mecanismo muito melhor, mais barato e mais simples para atingir todos esses objetivos.

- 7,13 Ainda é necessário marcar o texto com [o equivalente funcional de] barras de alteração e datas de revisão. Ainda é necessário um resumo eletrônico da mudança UEPS.
- 7,14 Parnas argumenta fortemente que o objetivo de todos vendo tudo é *totalmente errado*; as partes devem ser encapsuladas de forma que ninguém precise ou tenha permissão para ver as partes internas de qualquer parte que não seja a sua, mas deva ver apenas as interfaces.
- 7,15 A proposta de Parnas é uma receita para o desastre. [*I foram bastante convencidos do contrário por Parnas, e mudei totalmente de ideia.*]

Organização

- 7.16 O objetivo da organização é reduzir a quantidade de comunicação e coordenação necessárias.
- 7.17 Organização incorpora *divisão de trabalho* e *especialização da função* a fim de evitar a comunicação.
- 7.18 A organização da árvore convencional reflete o *autoridade* princípio da estrutura de que nenhuma pessoa pode servir a dois senhores.
- 7.19 o *comunicação* a estrutura em uma organização é uma rede, não uma árvore, portanto, todos os tipos de mecanismos de organização especiais ("linhas pontilhadas") devem ser concebidos para superar as deficiências de comunicação da organização estruturada em árvore.
- 7.20 Cada subprojeto tem duas funções de liderança a serem preenchidas, a de *produtor* e o do *diretor técnico*, ou *arquiteto*. As funções das duas funções são bastante distintas e requerem talentos diferentes.
- 7.21 Qualquer um dos três relacionamentos entre as duas funções pode ser bastante eficaz:
- O produtor e o diretor podem ser os mesmos.
 - O produtor pode ser o chefe, e o diretor, o braço direito do produtor.
 - O diretor pode ser o chefe e o produtor o braço direito do diretor.

Capítulo 8. Chamando o tiro

- 8,1 Não se pode estimar com precisão o esforço total ou o cronograma de um projeto de programação simplesmente estimando o tempo de codificação e multiplicando por fatores para as outras partes da tarefa.
- 8,2 Os dados para a construção de pequenos sistemas isolados não são aplicáveis a projetos de sistemas de programação.
- 8,3 Os aumentos de programação ocorrem como uma potência do tamanho do programa.
- 8,4 Alguns estudos publicados mostram que o expoente é cerca de 1,5. [*Os dados de Boehm não concordam com isso, mas variam de 1,05 a 1,2.*] ¹
- 8,5 Os dados ICL de Portman mostram programadores em tempo integral aplicando apenas cerca de 50% de seu tempo em programação e depuração, em comparação com outras tarefas do tipo overhead.
- 8,6 Os dados da IBM de Aron mostram que a produtividade varia de 1,5 K linhas de código (KLOC) por homem-ano a 10 KLOC / homem-ano em função do número de interações entre as partes do sistema.
- 8,7 Os dados do Harr's Bell Labs mostram produtividades no trabalho do tipo sistema operacional para executar cerca de 0,6 KLOC / homem-ano e no trabalho do tipo compilador cerca de 2,2 KLOC / homem-ano para produtos acabados.
- 8,8 Os dados do OS / 360 de Brooks concordam com os de Harr: 0,6-0,8 KLOC / homem-ano em sistemas operacionais e 2-3 KLOC / homem-ano em compiladores.
- 8,9 Os dados do MIT Project MULTICS de Corbatd mostraram uma produtividade de 1,2 KLOC / homem-ano em uma mistura de sistemas operacionais e compiladores, mas essas são linhas de código PL / I, enquanto todos os outros dados são linhas de código assembler!
- 8,10 A produtividade parece constante em termos de afirmações elementares.
- 8,11 A produtividade da programação pode ser aumentada em até cinco vezes quando uma linguagem de alto nível adequada é usada.

Capítulo 9. Dez libras em um saco de cinco libras

- 9,1 Além do tempo de execução, o *espaço de memória* ocupada por um programa é um custo principal. Isso é especialmente verdadeiro para sistemas operacionais, onde muitos deles residem o tempo todo. Mesmo
- 9,2 assim, o dinheiro gasto em memória para a residência do programa pode render um valor funcional por dólar muito bom,[^] melhor do que outras formas de investir em configuração. O tamanho do programa não é ruim; tamanho desnecessário é.
- 9,3. O construtor de software deve definir metas de tamanho, controlar o tamanho e desenvolver técnicas de redução de tamanho, assim como o construtor de hardware faz para os componentes.
- 9,4 Os orçamentos de tamanho devem ser explícitos não apenas sobre o tamanho residente, mas também sobre os acessos ao disco ocasionados por buscas de programas.
- 9,5 Os orçamentos de tamanho devem estar vinculados às atribuições de funções; defina exatamente o que um módulo deve fazer quando você especifica o quanto grande ele deve ser.
- 9,6 Em equipes grandes, as subequipes tendem a subotimizar para cumprir suas próprias metas, em vez de pensar no efeito total sobre o usuário. Essa falha na orientação é um grande risco em grandes projetos.
- 9,7 Durante toda a implementação, os arquitetos de sistema devem manter vigilância constante para garantir a integridade contínua do sistema.
- 9,8 Promover uma atitude de sistema total e orientada para o usuário pode muito bem ser a função mais importante do gerenciador de programação.
- 9,9 Uma decisão política inicial é decidir quanto refinada será a escolha de opções do usuário, uma vez que empacotá-las em grupos economiza espaço de memória [e freqüentemente custos de marketing]. O
- 9,10 tamanho da área transitória, portanto a quantidade de programa por busca de disco, é uma decisão crucial, uma vez que o desempenho é uma função superlinear desse tamanho. [Toda essa decisão ficou obsoleta, primeiro pela memória virtual, depois

por memória real barata. Os usuários agora normalmente compram memória real suficiente para armazenar todo o código dos principais aplicativos.]

- 9.11 Para fazer boas compensações espaço-tempo, uma equipe precisa ser treinada nas técnicas de programação peculiares a uma linguagem ou máquina particular, especialmente uma nova.
- 9.12 A programação tem uma tecnologia e todo projeto precisa de uma biblioteca de componentes padrão.
- 9.13 Bibliotecas de programas devem ter duas versões de cada componente, a rápida e a comprimida. [Isso parece obsoleto hoje.]
- 9.14 Programas enxutos, sobressalentes e rápidos são quase sempre o resultado de *descoberta estratégica*, em vez de inteligência prática.
- 9.15 Freqüentemente, tal avanço será um novo *algoritmo*.
- 9.16 Mais frequentemente, o avanço virá ao refazer o *representação* dos dados ou tabelas. *A representação é a essência da programação.*

Capítulo 10. A Hipótese Documentária

- 10.1 "A hipótese: em meio a uma lavagem de papel, um pequeno número de documentos torna-se os eixos críticos em torno dos quais gira todo gerenciamento de projeto. Essas são as principais ferramentas pessoais do gerente."
- 10.2 Para um projeto de desenvolvimento de computador, os documentos críticos são os objetivos, manual, cronograma, orçamento, organograma, alocação de espaço físico e a estimativa, previsão e preços da própria máquina.
- 10.3 Para um departamento de universidade, os documentos críticos são semelhantes: os objetivos, requisitos de graduação, descrições de cursos, propostas de pesquisa, programação de aulas e plano de ensino, orçamento, alocação de espaço físico e designações de funcionários e assistentes de pós-graduação.
- 10.4 Para um projeto de software, as necessidades são as mesmas: os objetivos, manual do usuário, documentação interna, cronograma, orçamento, organograma e alocação de espaço físico.
- 10.5 Mesmo em um projeto pequeno, portanto, o gerente deve

- desde o início formalizar esse conjunto de documentos. A
- 10,6 preparação de cada documento deste pequeno conjunto concentra o pensamento e cristaliza a discussão. O ato de escrever requer centenas de mini-decisões, e é a existência delas que distingue as políticas claras e exatas das indefinidas.
- 10,7 Manter cada documento crítico fornece uma vigilância de status e mecanismo de alerta.
- 10,8 Cada documento serve como uma lista de verificação e um
- 10,9 banco de dados. O trabalho fundamental do gerente de projeto é manter todos na mesma direção.
- 10,10 A principal tarefa diária do gerente de projetos é a comunicação, não a tomada de decisões; os documentos comunicam os planos e decisões a toda a equipe.
- 10,11 Apenas uma pequena parte do tempo de um gerente de projeto técnico - talvez 20% - é gasto em tarefas nas quais ele precisa de informações externas.
- 10,12 Por esse motivo, o conceito de mercado de um "sistema de gerenciamento de informação total" para apoiar os executivos não se baseia em um modelo válido de comportamento executivo.

Capítulo 11. Planeje jogar fora

- 11,1 Engenheiros químicos aprenderam a não levar um processo da bancada do laboratório para a fábrica em uma única etapa, mas a construir um *planta piloto* para dar experiência em aumentar as quantidades e operar em ambientes não protetores.
- 11,2 Essa etapa intermediária é igualmente necessária para produtos de programação, mas os engenheiros de software ainda não testam rotineiramente em campo um sistema piloto antes de entregar o produto real. [Isso agora se tornou uma prática comum, com uma versão beta. Isso não é o mesmo que um protótipo com função limitada, uma versão alfa, que eu também defenderia.]
- 11,3 Para a maioria dos projetos, o primeiro sistema construído quase não pode ser usado: muito lento, muito grande, muito difícil de usar ou todos os três.

- 11.4 O descarte e o redesenho podem ser feitos de uma só vez, ou pedaço por pedaço, mas *será feito*.
- 11.5 Entregar o primeiro sistema, o descartável, aos usuários ganhará tempo, mas apenas ao custo da agonia para o usuário, distração para os construtores que o apoiam enquanto fazem o redesenho e uma má reputação do produto que será difícil de viver para baixo.
- 11.6 Portanto, *planeje jogar um fora; você vai, de qualquer maneira*.
- 11.7 "O programador fornece a satisfação de uma necessidade do usuário ao invés de qualquer produto tangível." (Cosgrove)
- 11.8 Tanto a necessidade real quanto a percepção do usuário dessa necessidade *mudança* à medida que os programas são criados, testados e usados.
- 11.9 A tratabilidade e a invisibilidade do produto de software expõem seus construtores (excepcionalmente) a mudanças perpétuas nos requisitos.
- 11.10 Algumas mudanças válidas nos objetivos (e nas estratégias de desenvolvimento) são inevitáveis, e é melhor estar preparado para elas do que presumir que elas não virão.
- 11.11 As técnicas de planejamento de um produto de software para mudança, especialmente a programação estruturada com documentação cuidadosa da interface do módulo, são bem conhecidas, mas não uniformemente praticadas. Também ajuda a usar técnicas baseadas em tabelas sempre que possível. [Os custos e tamanhos de memória modernos tornam essas técnicas cada vez melhores.]
- 11.12 Use linguagem de alto nível, operações de tempo de compilação, incorporações de declarações por referência e técnicas de autodocumentação para reduzir erros induzidos por mudanças.
- 11.13 Quantifique as mudanças em versões numeradas bem definidas. [Agora, prática padrão.]

Planeje a organização para a mudança

- 11.14 A relutância do programador em documentar designs não vem tanto da preguiça, mas da hesitação em assumir a defesa de decisões que o designer sabe que são provisórias. (Cosgrove)

- 11.15 Estruturar uma organização para mudanças é muito mais difícil do que projetar um sistema para mudanças.
- 11.16 O chefe do projeto deve trabalhar para manter os gerentes e o pessoal técnico tão intercambiáveis quanto seus talentos permitirem; em particular, deseja-se ser capaz de mover as pessoas facilmente entre funções técnicas e gerenciais.
- 11.17 As barreiras para uma organização de escada dupla eficaz são sociológicas e devem ser combatidas com vigilância e energia constantes.
- 11.18 É fácil estabelecer escalas salariais correspondentes para os degraus correspondentes em uma escada dupla, mas requer fortes medidas proativas para dar-lhes o prestígio correspondente: escritórios iguais, serviços de apoio iguais, ações de gestão compensatórias.
- 11.19 A organização de uma equipe cirúrgica é um ataque radical a todos os aspectos desse problema. É realmente a resposta de longo prazo para o problema da organização flexível.

Duas etapas à frente e uma etapa atrás - Manutenção do programa

- 11.20 A manutenção do programa é fundamentalmente diferente da manutenção do hardware; consiste principalmente em alterações que reparam defeitos de design, adicionam funções incrementais ou se adaptam a alterações no ambiente de uso ou na configuração.
- 11.21 O custo total de vida de manutenção de um programa amplamente utilizado é normalmente de 40 por cento ou mais do custo de desenvolvê-lo.
- 11.22 O custo de manutenção é fortemente afetado pelo número de usuários. Mais usuários encontram mais bugs.
- 11.23 Campbell aponta uma curva interessante de queda e subida em bugs por mês ao longo da vida de um produto.
- 11.24 Consertar um defeito tem uma chance substancial (20 a 50 por cento) de introduzir outro.
- 11.25 Após cada correção, deve-se executar todo o banco de casos de teste executados anteriormente em um sistema para garantir que ele não

foi danificado de forma obscura.

- 11.26 Métodos de projetar programas de modo a eliminar ou pelo menos iluminar os efeitos colaterais podem ter uma grande recompensa em custos de manutenção.
- 11.27 O mesmo pode acontecer com os métodos de implementação de projetos com menos pessoas, menos interfaces e menos bugs.

Um passo à frente e um passo atrás - A entropia do sistema aumenta ao longo da vida

- 11.28 Lehman e Belady descobrem que o número total de módulos aumenta linearmente com o número da versão de um grande sistema operacional (OS / 360), mas que o número de módulos afetados aumenta exponencialmente com o número da versão.
- 11.29 Todos os reparos tendem a destruir a estrutura, a aumentar a entropia e a desordem de um sistema. Mesmo a manutenção de programa mais habilidosa apenas atrasa a subsidência do programa em um caos incontrolável, a partir do qual deve haver um redesenho básico.
[Muitas das necessidades reais de atualização de um programa, como o desempenho, atacam especialmente seus limites estruturais internos. Muitas vezes, os limites originais ocasionaram as deficiências que surgiram mais tarde.]

Capítulo 12. Ferramentas Sharp

- 12.1 O gerente de um projeto precisa estabelecer uma filosofia e reservar recursos para a construção de ferramentas comuns e, ao mesmo tempo, reconhecer a necessidade de ferramentas personalizadas.
- 12.2 As equipes que criam sistemas operacionais precisam de uma máquina-alvo própria para depurar; ele precisa de memória máxima em vez de velocidade máxima e um programador de sistema para manter o software padrão atualizado e em condições de uso.
- 12.3 A máquina de depuração, ou seu software, também precisa ser instrumentada, de modo que contagens e medições de todos

- 12,4 tipos de parâmetros de programa podem ser feitos automaticamente. O requisito para o uso da máquina-alvo tem uma curva de crescimento peculiar: baixa atividade seguida de crescimento explosivo e, em seguida, estabilização.
- 12,5 A depuração do sistema, como a astronomia, sempre foi feita principalmente à noite.
- 12,6 Alocar blocos substanciais de tempo da máquina-alvo para uma subequipe por vez provou ser a melhor maneira de programar, muito melhor do que intercalar o subequipe, apesar da teoria.
- 12,7 Este método preferido de escalonamento de computadores escassos por blocos sobreviveu 20 anos [em 1975] de mudanças tecnológicas porque é mais produtivo. [Ainda é, em 1995]. Se um computador de destino é novo, é necessário um simulador lógico para ele. Um entende *mais cedo*, e fornece um *confiável*/veículo de depuração, mesmo depois de ter *para* máquina real. A biblioteca do programa Amaster deve ser dividida em (1) um conjunto de brinquedos individuais, (2) uma sub-biblioteca de integração do sistema, atualmente em teste do sistema, e (3) uma versão lançada. A separação formal e a progressão dão o controle.
- 12.10 A ferramenta que economiza mais trabalho em um projeto de programação é provavelmente um sistema de edição de texto.
- 12.11 A voluminosidade na documentação do sistema realmente introduz um novo tipo de incompreensibilidade [ver Unix, por exemplo], mas é muito preferível à severa subdocumentação que é tão comum.
- 12.12 Construa um simulador de desempenho, de fora para dentro, de cima para baixo. Comece bem cedo. Ouça quando ele fala.

Linguagem de alto nível

- 12.13 Apenas a preguiça e a inércia impedem a adoção universal de linguagem de alto nível e programação interativa. [E hoje eles foram universalmente adotados]
- 12.14 A linguagem de alto nível melhora não apenas a produtividade, mas também a depuração; menos bugs e mais fácil de encontrar
- 12.15 As objeções clássicas de função, espaço de código-objeto,

e a velocidade do código-objeto tornou-se obsoleta com o avanço da linguagem e da tecnologia do compilador.

12.16 O único candidato razoável para a programação do sistema hoje é PL / I. [Não é mais verdade.]

Interactive Programming

12.17 Os sistemas interativos nunca substituirão os sistemas em lote para alguns aplicativos. [Ainda é verdade.]

12.18 A depuração é a parte difícil e lenta da programação do sistema, e o retorno lento é a ruína da depuração.

12.19 Evidências limitadas mostram que a programação interativa pelo menos dobra a produtividade na programação do sistema.

Capítulo 13. O Todo e as Partes

13.1 O esforço arquitetônico detalhado e meticuloso implícito nos Capítulos 4, 5 e 6 não apenas torna o produto mais fácil de usar, mas também torna mais fácil construir e reduz o número de bugs do sistema que precisam ser encontrados.

13.2 Vyssotsky diz "Muitas, muitas falhas dizem respeito exatamente àqueles aspectos que nunca foram totalmente especificados."

13.3 Muito antes de qualquer código em si, a especificação deve ser entregue a um grupo de teste externo para ser examinado quanto à integridade e clareza. Os próprios desenvolvedores não podem fazer isso. (Vyssotsky)

13.4 "O projeto de cima para baixo de Wirth [por refinamento gradual] é a nova formalização de programação mais importante da década [1965-1975]."

13.5 Wirth defende o uso de uma notação de alto nível possível em cada etapa.

13.6 Um bom design de cima para baixo evita bugs de quatro maneiras.

13.7 Às vezes, é preciso voltar, descartar um nível alto e começar de novo.

13.8 Programação estruturada, projetar programas cujas estruturas de controle consistem apenas em um conjunto especificado que governa blocos de código (versus ramificações diversas), é um

- boa maneira de evitar bugs e é a maneira certa de pensar. Os
- 13.9 resultados experimentais de Gold mostram três vezes mais progresso na primeira interação de uma sessão de depuração interativa do que nas interações subsequentes. Ainda vale a pena planejar a depuração cuidadosamente antes de se conectar. [Eu penso isso *ainda faz*, em 1995.]
- 13.10 Acho que o uso adequado de um bom sistema [depuração interativa de resposta rápida] requer duas horas na mesa para cada sessão de duas horas na máquina: uma hora para varrer e documentar após a sessão e outra para planejar mudanças e testes para a próxima vez.
- 13.11 A depuração do sistema (em contraste com a depuração do componente) levará mais tempo do que o esperado.
- 13.12 A dificuldade de depuração do sistema justifica uma abordagem totalmente sistemática e planejada.
- 13.13 Deve-se começar a depuração do sistema somente depois que as peças parecerem funcionar (versus-parafusar-junto-e-tentar para eliminar os bugs da interface; e versus começar a depuração do sistema quando os bugs do componente são totalmente conhecidos, mas não corrigidos.) [Isso é especialmente verdadeiro para as equipes.]
- 13.14 Vale a pena construir muitos scaffolds de depuração e código de teste, talvez até 50% do valor do produto que está sendo depurado.
- 13.15 Deve-se controlar e documentar as mudanças e versões, com os membros da equipe trabalhando em cópias do cercadinho.
- 13.16 Adicione um componente por vez durante a depuração do sistema.
- 13.17 Lehman e Belady oferecem evidências de que os quanta de mudança devem ser grandes e raros, ou então muito pequenos e frequentes. Este último está mais sujeito à instabilidade. [Uma equipe da Microsoft faz com que pequenos quanta frequentes funcionem. O sistema de cultivo é reconstruído todas as noites.]

Capítulo 14. Chocando uma catástrofe

- 14.1 "Como um projeto chega a atrasar um ano? ... Um dia de cada vez."

- 14.2 O deslizamento do cronograma do dia a dia é mais difícil de reconhecer, mais difícil de prevenir e mais difícil de compensar do que calamidades.
- 14.3 O primeiro passo para controlar um grande projeto em um cronograma apertado é *tenho* uma programação, composta de marcos e datas para eles.
- 14.4 Os marcos devem ser eventos concretos, específicos e mensuráveis, definidos com nitidez de lâmina de faca.
- 14.5 Um programador raramente mentirá sobre o progresso do marco, se o marco for tão preciso que ele não pode se enganar.
- 14.6 Estudos de estimativa de comportamento por empreiteiros do governo em grandes projetos mostram que as estimativas de tempo de atividade revisadas cuidadosamente a cada duas semanas não mudam significativamente com a aproximação do tempo de início, que durante a atividade as superestimativas diminuem constantemente; e essa *subestima* não mude até cerca de três semanas antes da conclusão programada.
- 14.7 O deslizamento crônico do cronograma é um assassino moral. [Jim McCarthy, da Microsoft, diz: "Se você perder um prazo, certifique-se de cumprir o próximo." ²⁾]
- 14.8 *Labuta* é essencial para grandes equipes de programação, assim como para grandes equipes de beisebol.
- 14.9 Não há substituto para um cronograma de caminho crítico para permitir que alguém diga quais deslizes importam quanto.
- 14.10 A preparação de um gráfico de caminho crítico é a parte mais valiosa de seu uso, uma vez que traçar a rede, identificar as dependências e estimar os segmentos exige um grande planejamento muito específico no início de um projeto.
- 14.11 O primeiro gráfico é sempre terrível, e alguém inventa e inventa ao fazer o próximo.
- 14.12 Um gráfico de caminho crítico responde à desculpa desmoralizante: "A outra peça está atrasada, de qualquer maneira."
- 14.13 Todo chefe precisa de informações de exceção que requerem ação e um quadro de status para educação e aviso prévio à distância.
- 14.14 Obter o status é difícil, uma vez que os gerentes subordinados

tem todos os motivos para não compartilhá-lo.

- 14.15 Por má ação, um chefe pode garantir silenciar a divulgação total do status; inversamente, separar cuidadosamente os relatórios de status e aceitá-los sem pânico ou preempção incentivará o relato honesto.
- 14.16 Deve-se ter técnicas de revisão pelas quais o verdadeiro status se torna conhecido por todos os jogadores. Para este propósito, um cronograma de marcos e um documento de conclusão são a chave.
- 14.17 Vyssotsky: "Achei útil incluir datas 'programadas' (datas do chefe) e 'estimadas' (datas do gerente de nível mais baixo) no relatório de marcos. O gerente de projeto tem que manter seus dedos longe das datas estimadas."
- 14.18 Um pequeno *Planos e controles* A equipe que mantém o relatório de marcos é inestimável para um grande projeto.

Capítulo 15. A outra face

- 15.1 Para o produto do programa, a outra face para o usuário, a documentação, é tão importante quanto a face para a máquina.
- 15.2 Mesmo para o mais privado dos programas, a documentação em prosa é necessária, pois a memória falhará com o autor-usuário.
- 15.3 Professores e gerentes em geral falharam em incutir nos programadores uma atitude sobre a documentação que irá inspirar para o resto da vida, superando a preguiça e a pressão de cronograma.
- 15.4 Esta falha não se deve tanto à falta de zelo ou eloquência, mas a uma falha em mostrar *Como as* para documentar de forma eficaz e econômica.
- 15.5 A maioria da documentação falha em fornecer muito pouco *visão global*.
Afaste-se e amplie lentamente.
- 15.6 A documentação crítica do usuário deve ser elaborada antes da construção do programa, pois ela incorpora decisões básicas de planejamento. Deve descrever nove coisas (veja o capítulo).
- 15.7 Um programa deve ser enviado com alguns casos de teste, alguns

para dados de entrada válidos, alguns para dados de entrada limítrofes e alguns para dados de entrada claramente inválidos.

15.8 A documentação dos internos do programa, para quem deve modificá-lo, também exige uma visão geral em prosa, que deve conter cinco tipos de coisas (veja o capítulo).

15.9 O fluxograma é uma peça completamente exagerada da documentação do programa; o fluxograma detalhado passo a passo é um incômodo, obsoleto por *escrito* linguagens de alto nível. (Um fluxograma é um *diagramado* linguagem de alto nível.)

15.10 Poucos programas precisam de mais do que um fluxograma de uma página, se tanto. [Os requisitos de documentação MILSPEC estão realmente errados neste ponto.]

15.11 De fato, é necessário um gráfico da estrutura do programa, que não precisa dos padrões de fluxograma ANSI.

15.12 Para manter a documentação mantida, é crucial que seja incorporada no programa de origem, ao invés de mantida como um documento separado.

15.13 Três noções são fundamentais para minimizar a carga de documentação:

- Use partes do programa que devem estar lá de qualquer maneira, como nomes e declarações, para transportar o máximo de documentação possível.
- Use espaço e formato para mostrar subordinação e aninhamento e para melhorar a legibilidade.
- Insira a documentação necessária em prosa no programa como parágrafos de comentários, especialmente como cabeçalhos de módulos.

15.14 Na documentação para uso por modificadores de programa, diga *porque as coisas são como são, e não apenas como são.*

Propósito é a chave para a compreensão; mesmo a sintaxe da linguagem de alto nível não transmite de forma alguma um propósito.

15.15 Técnicas de programação autodocumentáveis encontram seu maior uso e poder em linguagens de alto nível usadas com sistemas on-line, que são as ferramentas *deve* estar usando.

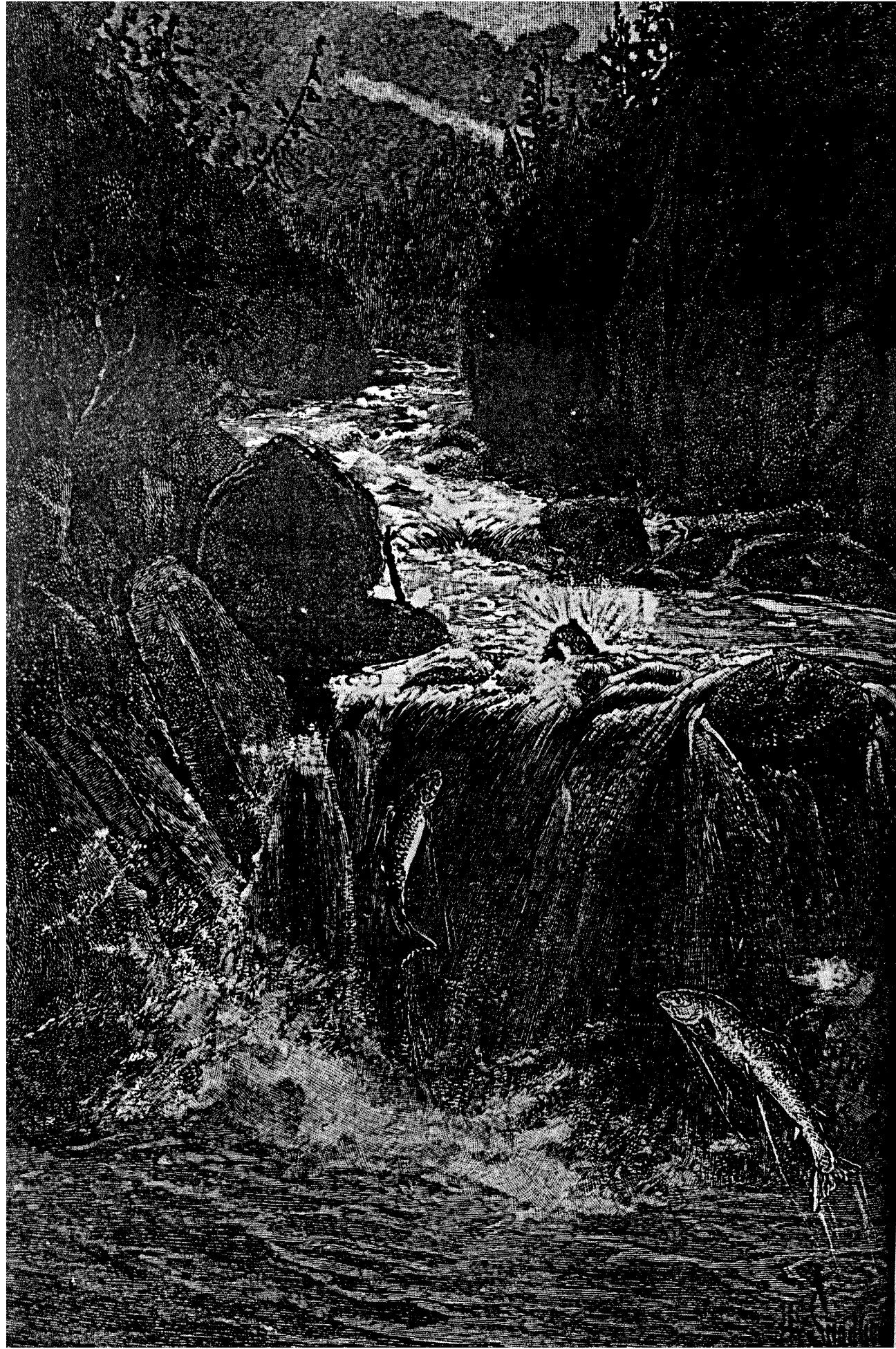
Epílogo Original

- EI Os sistemas de software são talvez os mais intrincados e complexos (em termos de número de tipos distintos de peças) das coisas que a humanidade faz.
- E.2 O alcatrão da engenharia de software continuará a ser difícil por muito tempo.

19

O Mítico Homem-Mês

depois de 20 anos



19

O Mítico Homem-Mês

depois de 20 anos

Não conheço maneira de julgar o futuro senão pelo passado.

PATRICK HENRY

Você nunca pode planejar o futuro pelo passado.

EDMUND BURKE

Atirando nas corredeiras
TheBettmanArchive

Por que existe uma edição do vigésimo aniversário?

O avião zumbiu noite adentro em direção a LaGuardia. Nuvens e escuridão velaram todos os pontos turísticos interessantes. O documento que eu estava estudando era trivial. Eu não estava, entretanto, entediado. O estranho sentado ao meu lado estava lendo *O Mítico Homem-Mês*, e eu estava esperando para ver se por palavra ou sinal ele reagiria. Finalmente, enquanto taxiamos em direção ao portão, não pude esperar mais:

"Como está esse livro? Você o recomenda?"

"Hmph! Nada nele que eu já não soubesse."

Decidi não me apresentar.

Por que você *O Mítico Homem-Mês* persistiu? Por que ainda é considerado relevante para a prática de software hoje? Por que ele tem leitores fora da comunidade de engenharia de software, gerando comentários, citações e correspondência de advogados, médicos, psicólogos, sociólogos, bem como de pessoal de software? Como um livro escrito há 20 anos sobre uma experiência de construção de software há 30 anos ainda pode ser relevante, muito menos útil?

Uma explicação que às vezes se ouve é que a disciplina de desenvolvimento de software não avançou normalmente ou de maneira adequada. Essa visão é freqüentemente apoiada pelo contraste da produtividade de desenvolvimento de software de computador com a produtividade de fabricação de hardware de computador, que se multiplicou pelo menos mil vezes ao longo das duas décadas. Como o Capítulo 16 explica, a anomalia não é que o software tenha sido tão lento em seu progresso, mas sim que a tecnologia do computador explodiu de uma maneira sem paralelo na história humana. Em geral, isso vem da transição gradual da manufatura de computadores de uma indústria de montagem para uma indústria de processo, de uma manufatura intensiva em trabalho para uma manufatura intensiva em capital. O desenvolvimento de hardware e software, em contraste com a manufatura, permanece inherentemente intensivo em mão de obra.

Uma segunda explicação frequentemente avançada é que *O Mítico Homem-Mês* é apenas incidentalmente sobre software, mas principalmente sobre como as pessoas em equipes fazem as coisas. Certamente há alguma verdade nisso; o prefácio da edição de 1975 diz que a gestão

um projeto de software é mais parecido com outro gerenciamento do que a maioria dos programadores inicialmente acredita. Eu ainda acredito que isso seja verdade. A história humana é um drama no qual as histórias permanecem as mesmas, os roteiros dessas histórias mudam lentamente com a evolução das culturas e os cenários mudam o tempo todo. É assim que vemos nosso eu do século XX espelhado em Shakespeare, Homero e na Bíblia. Então, na medida *o MM-M* é sobre pessoas e equipes, a obsolescência deve ser lenta.

Seja qual for o motivo, os leitores continuam a comprar o livro e continuam a me enviar comentários muito apreciados. Hoje em dia, muitas vezes me perguntam: "O que você acha que estava errado quando foi escrito? O que agora está obsoleto? O que é realmente novo no mundo da engenharia de software?" Essas questões bem distintas são todas justas e irei abordá-las da melhor maneira que puder. Não nessa ordem, porém, mas em grupos de tópicos. Primeiro, vamos considerar o que estava certo quando escrito e ainda é.

O argumento central: integridade conceitual e o arquiteto

Integridade conceitual. Um produto de programação limpo e elegante deve apresentar a cada um de seus usuários um modelo mental coerente da aplicação, das estratégias para fazer a aplicação e das táticas de interface do usuário a serem usadas na especificação de ações e parâmetros. A integridade conceitual do produto, conforme percebida pelo usuário, é o fator mais importante na facilidade de uso. (Existem outros fatores, é claro. A uniformidade da interface do usuário do Macintosh em todos os aplicativos é um exemplo importante. Além disso, é possível construir interfaces coerentes que, no entanto, são difíceis de remover. Considere o MS-DOS.)

Existem muitos exemplos de produtos de software elegantes projetados por uma única mente ou por um par. A maioria das obras puramente intelectuais, como livros ou composições musicais, são assim produzidas. Os processos de desenvolvimento de produtos em muitos setores não podem, entretanto, permitir essa abordagem direta à integridade conceitual. As pressões competitivas forçam a urgência; em muitos modernos

tecnologias o produto final é bastante complexo e o design requer inherentemente muitos homens-meses de esforço. Os produtos de software são complexos e extremamente competitivos em termos de prazos.

Qualquer produto que seja suficientemente grande ou urgente para exigir o esforço de muitas mentes, portanto, encontra uma dificuldade peculiar: o resultado deve ser conceitualmente coerente com a mente única do usuário e, ao mesmo tempo, projetado por muitas mentes. Como organizar os esforços de design de forma a atingir tal integridade conceptual? Esta é a questão central abordada por *O MM-M.*

Uma de suas teses é que gerenciar grandes projetos de programação é qualitativamente diferente de gerenciar pequenos, apenas por causa do número de mentes envolvidas. Ações de gerenciamento deliberadas, e mesmo heróicas, são necessárias para alcançar a coerência.

O arquiteto. Eu argumento nos Capítulos 4 a 7 que a ação mais importante é o comissionamento de uma mente para ser o produto *arquiteto*, quem é responsável pela integridade conceitual de todos os aspectos do produto perceptíveis pelo usuário. O arquiteto forma e possui o modelo mental público do produto que será usado para explicar seu uso ao usuário. Isso inclui a especificação detalhada de todas as suas funções e os meios para invocá-la e controlá-la. O arquiteto também é o agente do usuário, representando com conhecimento o interesse do usuário nas compensações inevitáveis entre função, desempenho, tamanho, custo e cronograma. Essa função é de tempo integral e apenas nas equipes menores pode ser combinada com a do gerente de equipe. O arquiteto é como o diretor e o gerente como o produtor de um filme.

Separação da arquitetura da implementação e realização. Para tornar a tarefa crucial do arquiteto mesmo concebível, é necessário separar a arquitetura, a definição do produto percebido pelo usuário, de sua implementação. Arquitetura versus implementação define um limite claro entre as partes da tarefa de design, e há muito trabalho em cada lado dela.

Recursão de arquitetos. Para produtos muito grandes, uma mente não pode fazer toda a arquitetura, mesmo depois que todas as questões de implementação foram separadas. Portanto, é necessário que o arquiteto mestre do sistema particione o sistema em subsistemas. Os limites do subsistema devem estar nos locais onde as interfaces entre os subsistemas são mínimas e mais fáceis de definir com rigor. Então, cada peça terá seu próprio arquiteto, que deve se reportar ao arquiteto mestre do sistema no que diz respeito à arquitetura. É claro que esse processo pode prosseguir recursivamente conforme necessário.

Hoje estou mais convencido do que nunca. Integridade conceitual é central para a qualidade do produto. Ter um arquiteto de sistema é o passo mais importante em direção à integridade conceitual. Esses princípios não se limitam de forma alguma aos sistemas de software, mas ao projeto de qualquer construção complexa, seja um computador, um avião, uma Iniciativa de Defesa Estratégica, um Sistema de Posicionamento Global. Depois de ensinar em um laboratório de engenharia de software mais de 20 vezes, passei a insistir que equipes de alunos de apenas quatro pessoas escolhessem um gerente e um arquiteto separado. Definir papéis distintos em equipes tão pequenas pode ser um pouco extremo, mas observei que funciona bem e contribui para o sucesso do design, mesmo para equipes pequenas.

O efeito do segundo sistema: Featuritis e adivinhação de frequência

Projetando para grandes conjuntos de usuários. Uma das consequências da revolução do computador pessoal é que cada vez mais, pelo menos na comunidade de processamento de dados corporativos, os pacotes de prateleira estão substituindo os aplicativos personalizados. Além disso, os pacotes de software padrão vendem centenas de milhares de cópias, ou até milhões. Os arquitetos de sistema de software fornecido por fornecedores de máquinas sempre tiveram que projetar para um conjunto de usuários grande e amorfo, em vez de para um único aplicativo definível em uma empresa. Muitos, muitos arquitetos de sistema agora enfrentam essa tarefa.

Paradoxalmente, é muito mais difícil projetar um

ferramenta de propósito do que projetar uma ferramenta de propósito especial, precisamente porque é preciso atribuir pesos às diferentes necessidades dos diversos usuários.

Featuritis. A tentação persistente para o arquiteto de uma ferramenta de uso geral, como uma planilha ou um processador de texto, é sobreregar o produto com recursos de utilidade marginal, em detrimento do desempenho e até mesmo da facilidade de uso. O apelo dos recursos propostos é evidente no início; a penalidade de desempenho é evidente apenas à medida que o teste do sistema prossegue. A perda de facilidade de uso surge insidiosamente, à medida que recursos são adicionados em pequenos incrementos e os manuais ficam cada vez mais grossos.¹

Para produtos de mercado de massa que sobrevivem e evoluem por muitas gerações, a tentação é especialmente forte. Milhões de clientes solicitam centenas de recursos; qualquer pedido é em si uma evidência de que "o mercado o exige". Freqüentemente, o arquiteto do sistema original conquistou grandes glórias, e a arquitetura está nas mãos de pessoas com menos experiência em representar o interesse geral do usuário em equilíbrio. Uma análise recente do Microsoft Word 6.0 diz "O Word 6.0 inclui recursos; a atualização fica mais lenta pela bagagem ... O Word 6.0 também é grande e lento." Ele nota com consternação que o Word 6.0 requer 4 MB de RAM e prossegue dizendo que a rica função adicionada significa que "até mesmo um Macintosh IIfx [é] apenas capaz de cumprir a tarefa do Word 6".²

Definindo o conjunto do usuário. Quanto maior e mais amorfo for o conjunto de usuários, mais necessário será defini-lo explicitamente se quisermos atingir a integridade conceitual. Cada membro da equipe de design certamente terá uma imagem mental implícita dos usuários, e a imagem de cada designer será diferente. Uma vez que a imagem do usuário do arquiteto, consciente ou inconscientemente, afeta todas as decisões arquitetônicas, é essencial para *para* equipe de design para chegar a uma única imagem compartilhada. E isso requer anotar os atributos do conjunto de usuários esperado, incluindo:

- * Quem eles são
- * O que eles precisam

* O que eles acham que precisam

* O que eles querem

Frequências. Para qualquer produto de software, qualquer um dos atributos do conjunto do usuário é na verdade uma distribuição, com muitos valores possíveis, cada um com sua própria frequência. Como o arquiteto vai chegar a essas frequências? Pesquisar essa população mal definida é uma proposta duvidosa e cara.³ Com o passar dos anos, me convenci de que um arquiteto deveria *Acho*, ou, se preferir, *postulado*, um conjunto completo de atributos e valores com suas frequências, a fim de desenvolver uma descrição completa, explícita e compartilhada do conjunto de usuários.

Muitos benefícios decorrem desse procedimento improvável. Primeiro, o processo de adivinhar cuidadosamente as frequências fará com que o arquiteto pense muito cuidadosamente sobre o conjunto de usuários esperado. Em segundo lugar, anotar as frequências irá sujeitá-las a debate, o que iluminará todos os participantes e trará à tona as diferenças nas imagens do usuário que os vários designers carregam. Terceiro, enumerar as frequências explicitamente ajuda a todos a reconhecer quais decisões dependem de quais propriedades do conjunto de usuários. Mesmo esse tipo de análise de sensibilidade informal é valioso. Quando descobrir que decisões muito importantes dependem de algum palpite particular, vale a pena o custo para estabelecer melhores estimativas para esse valor.⁴ Não tive a oportunidade de usá-lo, mas acho que seria muito útil.)

Para resumir: escreva suposições explícitas para os atributos do conjunto de usuários. *É muito melhor ser explícito e o homem errado ser vago.*

E quanto ao "Efeito do Segundo Sistema"? Um aluno perspicaz observou que *O Mítico Homem-Mês* recomendou uma receita para o desastre: planeje entregar a segunda versão de qualquer novo sistema (Capítulo 11), que o Capítulo 5 caracteriza como o mais

sistema perigoso que alguém já projeta. Eu tive que conceder a ele um "te peguei".

A contradição é mais linguística do que real. O "segundo" sistema descrito no Capítulo 5 é o segundo sistema em campo, o sistema subsequente que convida a funções adicionais e enfeites. O "segundo" sistema no Capítulo 11 é a segunda tentativa de construir o que deveria ser o primeiro sistema a ser colocado em campo. É construído sob todas as restrições de cronograma, talento e ignorância que caracterizam os novos projetos - as restrições que exercem uma disciplina de magreza.

O triunfo da interface WIMP

Um dos desenvolvimentos mais impressionantes em software durante as últimas duas décadas foi o triunfo do Windows, Ícones, Menus, interface de apontamento - ou WIMP para breve. É hoje tão familiar que dispensa descrição. Este conceito foi exibido publicamente pela primeira vez por Doug Englebart e sua equipe do Stanford Research Institute na Western Joint Computer Conference de 1968.⁵ De lá, as ideias foram para o Xerox Palo Alto Research Center, onde surgiram na estação de trabalho pessoal Alto desenvolvida por Bob Taylor e sua equipe. Eles foram escolhidos por Steve Jobs para o Apple Lisa, um computador lento demais para carregar seus empolgantes conceitos de facilidade de uso. Esses conceitos, em seguida, Jobs incorporou no Apple Macintosh comercialmente bem-sucedido em 1985. Eles foram posteriormente adotados no Microsoft Windows para o IBM PC e compatível. A versão para Mac será meu exemplo.⁶

Integridade conceitual por meio de uma metáfora. O WIMP é um excelente exemplo de uma interface de usuário que tem integridade conceitual, alcançada pela adoção de um modelo mental familiar, a metáfora da área de trabalho, e sua extensão consistente e cuidadosa para explorar uma implementação de computação gráfica. Por exemplo, a decisão cara, mas adequada, de sobrepor as janelas em vez de colocá-las lado a lado decorre diretamente da metáfora. A capacidade de dimensionar e moldar janelas é uma extensão consistente que dá ao usuário os novos poderes habilitados pelo meio gráfico de computador. Artigos em um

a mesa não pode ser dimensionada e moldada tão facilmente. Arrastar e soltar seguem diretamente da metáfora; selecionar ícones apontando com um cursor é um análogo direto de pegar coisas com a mão. Ícones e pastas aninhadas são análogos fiéis aos documentos da área de trabalho; a lata de lixo também. Os conceitos de recortar, copiar e colar refletem fielmente o que costumávamos fazer com documentos em desktops. A metáfora é seguida com tanta fidelidade e sua extensão é tão consistente que novos usuários são positivamente abalados pela ideia de arrastar o ícone de um disquete para a lixeira para ejectar o disco. Se a interface não fosse quase uniformemente consistente, essa inconsistência (muito ruim) não irritaria tanto.

Onde a interface WIMP é forçada a ir muito além da metáfora da área de trabalho? Mais notavelmente em dois aspectos: menus e manuseio. Ao trabalhar com um desktop real, um *faz* ações para documentos, em vez de dizer a alguém ou algo para fazê-los. E quando alguém diz a alguém para fazer uma ação, geralmente gera, em vez de selecionar, os comandos verbais imperativos orais ou escritos: "Por favor, arquive isto." "Por favor, encontre a correspondência anterior." "Por favor, envie isso para Mary cuidar."

Infelizmente, a interpretação confiável de comandos em inglês gerados de forma livre está além do atual estado da arte, sejam os comandos escritos ou falados. Portanto, os designers de interface tiveram duas etapas removidas da ação direta do usuário nos documentos. Eles sabiamente escolheram da área de trabalho usual seu único exemplo de seleção de comando - a ficha impressa, na qual o usuário seleciona entre um menu restrito de comandos cuja semântica é padronizada. Essa ideia eles estenderam a um menu horizontal de submenus suspensos verticais.

Expressões de comando e o problema dos dois cursores. Commandos são sentenças imperativas; eles sempre têm um verbo e geralmente têm um objeto direto. Para qualquer ação, é necessário especificar um verbo e um substantivo. A metáfora do apontamento diz que, para especificar duas coisas por vez, tenha dois cursores distintos na tela, cada um dirigido por um mouse separado - um na mão direita e outro

no lado esquerdo. Afinal, em um desktop físico, normalmente trabalhamos com as duas mãos. (Mas, muitas vezes uma mão está segurando as coisas fixas no lugar, o que acontece por padrão na área de trabalho do computador.) A mente certamente é capaz de operar com as duas mãos; usamos regularmente as duas mãos para digitar, dirigir, cozinhar. Infelizmente, fornecer um mouse já era um grande passo para os fabricantes de computadores pessoais; nenhum sistema comercial acomoda duas ações simultâneas do cursor do mouse, uma dirigida com cada mão.⁷

Os designers de interface aceitaram a realidade e projetaram para um mouse, adotando a convenção sintática apontada (*Selecione% s*) o substantivo primeiro. Um aponta para o verbo, um item do menu. Isso realmente oferece muita facilidade de uso. Enquanto vejo usuários, ou fitas de vídeo de usuários, ou traçados de computador dos movimentos do cursor, fico imediatamente surpreso que um cursor está tendo que fazer o trabalho de dois: escolher um objeto na parte da área de trabalho da janela; escolha um verbo na parte do menu; encontre ou reencontre um objeto na área de trabalho; novamente abra um menu (geralmente o mesmo) e escolha um verbo. Para frente e para trás, para frente e para trás o cursor vai, de espaço de dados para espaço de menu, cada vez descartando as informações úteis sobre onde estava da última vez neste espaço - um processo totalmente ineficiente.

Uma solução brilhante. Mesmo que a eletrônica e o software possam lidar prontamente com dois cursores ativos simultaneamente, há dificuldades de layout de espaço. A área de trabalho na metáfora WIMP realmente inclui uma máquina de escrever, e deve-se acomodar seu teclado real no espaço físico da área de trabalho real. Um teclado e dois mouse pads usam muito do espaço de alcance do braço. Bem, o problema do teclado pode ser transformado em uma oportunidade - por que não habilitar a operação eficiente com as duas mãos usando uma mão no teclado para especificar verbos e a outra mão no mouse para escolher substantivos. Agora o cursor permanece no espaço de dados, explorando a alta localidade de sucessivas escolhas de substantivos. Eficiência real, poder real do usuário.

Poder do usuário versus facilidade de uso. Essa solução, no entanto, revela o que torna os menus tão fáceis de usar para os novatos - os menus apresentam os verbos alternativos válidos em qualquer estado particular. Podemos comprar um pacote, trazê-lo para casa e começar a usá-lo sem consultar o manual, apenas sabendo para que o compramos e experimentando os diferentes verbos do menu.

Uma das questões mais difíceis que os arquitetos de software enfrentam é exatamente como equilibrar o poder do usuário e a facilidade de uso. Eles são projetados para operação simples para o novato ou o usuário ocasional, ou para energia eficiente para o usuário profissional? A resposta ideal é fornecer ambos, de forma conceitualmente coerente - o que é obtido na interface WIMP. Cada um dos verbos do menu de alta frequência tem equivalentes de tecla única + tecla de comando, escolhidos principalmente para que possam ser facilmente tocados como um único acorde com a mão esquerda. No Mac, por exemplo, a tecla de comando (|) está logo abaixo das teclas Z e X; portanto, as operações de frequência mais alta são codificadas como z, x, c, v, s.

Transição incremental de novato para usuário avançado. Este sistema duplo para especificar verbos de comando não apenas atende às necessidades de esforço de baixo aprendizado do novato e às necessidades de eficiência do usuário avançado, ele fornece a cada usuário uma transição suave entre os modos. As codificações das letras, chamadas *atalthos*, são mostrados nos menus ao lado dos verbos, para que um usuário em dúvida possa puxar para baixo o menu para verificar o equivalente da letra, em vez de apenas escolher o item do menu. Cada novato aprende primeiro os atalhos para suas próprias operações de alta frequência. Qualquer atalho sobre o qual ele tenha dúvidas pode tentar, uma vez que & z irá desfazer qualquer único erro. Como alternativa, ele pode verificar o menu para ver quais comandos são válidos. Os novatos puxarão muitos menus; usuários avançados, muito poucos; e os usuários intermediários precisarão apenas ocasionalmente escolher em um menu, pois cada um conhecerá os poucos atalhos que compõem a maioria de suas próprias operações. A maioria de nós, designers de software, está muito familiarizada com essa interface para apreciar totalmente sua elegância e poder.

O sucesso da incorporação direta como um dispositivo para reforçar a arquitetura. A interface do Mac é notável de outra maneira. Sem coerção, seus designers o tornaram uma interface padrão entre os aplicativos, incluindo a grande maioria que é escrita por terceiros. Assim, o usuário ganha coerência conceitual no nível da interface, não apenas dentro do software fornecido com a máquina, mas em todos os aplicativos.

Essa façanha os designers do Mac conseguiram ao construir a interface na memória somente leitura, de modo que seja mais fácil e rápido para os desenvolvedores usá-la do que construir suas próprias interfaces idiossincráticas. Esses incentivos naturais para uniformidade prevaleceram amplamente o suficiente para estabelecer uma *de fato* padrão. Os incentivos naturais foram ajudados por um total comprometimento da administração e muita persuasão da Apple. Os revisores independentes nas revistas de produtos, reconhecendo o imenso valor da integridade conceitual de aplicação cruzada, também complementaram os incentivos naturais criticando impiedosamente os produtos que não estão em conformidade.

Este é um excelente exemplo da técnica, recomendada no Capítulo 6, de alcançar uniformidade encorajando outros a incorporar diretamente o código de alguém em seus produtos, em vez de tentar fazer com que construam seu próprio software de acordo com as especificações de alguém.

O destino do WIMP: Obsolescência. Apesar de suas excelências, eu ex- considere a interface WIMP como uma relíquia histórica em uma geração. Apontar ainda será a maneira de expressar substantivos enquanto comandamos nossas máquinas; a fala é certamente a maneira certa de expressar os verbos. Ferramentas como o Voice Navigator para Mac e Dragon para PC já oferecem esse recurso.

Não construa um para jogar fora - o modelo de cachoeira está errado!

A inesquecível imagem de Galloping Gertie, a Tacoma Narrows Bridge, abre o Capítulo 11, que recomenda radicalmente:

"Planeje jogar um fora; você vai, de qualquer maneira." Agora percebo que isso está errado, não porque seja muito radical, mas porque é muito simplista.

O maior erro no conceito de "Construir um para jogar fora" é que ele pressupõe implicitamente o modelo clássico sequencial ou em cascata de construção de software. O modelo deriva de um layout de gráfico de Gantt de um processo em estágios e geralmente é desenhado como na Figura 19.1. Winton Royce melhorou o modelo sequencial em um jornal clássico de 1970 ao fornecer

- * Algum feedback de um estágio para seus predecessores
- * Limitar o feedback apenas ao estágio imediatamente anterior, de modo a conter o custo e o atraso de programação que ele ocasiona.

Eu tenho precedido o MM-M em aconselhar os construtores a "construí-lo duas vezes".⁸ O Capítulo 11 não é o único manchado pelo modelo em cascata sequencial; ele percorre todo o livro, começando com a regra de agendamento no Capítulo 2. Essa regra aloca 1/3 do cronograma para planejamento, 1/6 para codificação, 1/4 para teste de componente e 1/4 para teste de sistema.

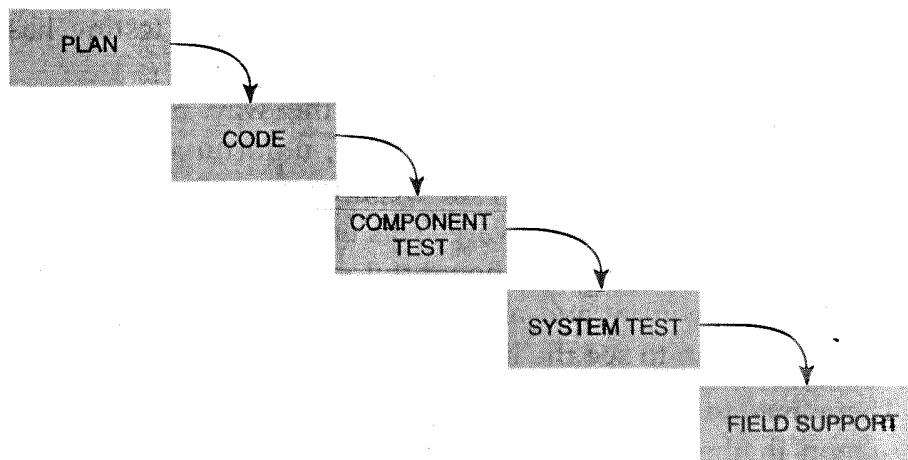


Fig. 19.1 Modelo em cascata de construção de software

A falácia básica do modelo em cascata é que ele assume que um projeto passa pelo processo *onze*, que a arquitetura é excelente e fácil de usar, o design de implementação é sólido e a realização pode ser corrigida à medida que o teste prossegue. Outra maneira de dizer isso é que o modelo em cascata presume que os erros estarão todos na realização e, portanto, seu reparo pode ser intercalado suavemente com testes de componentes e sistemas.

"Planeje jogar um fora" realmente ataca essa falácia de frente. Não é o diagnóstico que está errado; é o remédio. Bem, sugeri que se pudesse descartar e redesenhar o primeiro sistema, peça por peça, em vez de em um bloco. Até certo ponto, isso é bom, mas não consegue chegar à raiz do problema. O modelo em cascata testa o sistema e, portanto, por implicação *do utilizador* teste, no final do processo de construção. Assim, pode-se encontrar constrangimentos impossíveis para os usuários, ou desempenho inaceitável, ou suscetibilidade perigosa a erro ou malícia do usuário, somente após investir na construção completa. Para ter certeza, o escrutínio do teste Alpha das especificações visa encontrar tais falhas antecipadamente, mas não há substituto para usuários ativos.

A segunda falácia do modelo em cascata é que ele assume que se constrói um sistema inteiro de uma vez, combinando as peças para um teste de sistema ponta a ponta após todo o design de implementação, a maior parte da codificação e muitos dos testes de componentes foi feito.

O modelo em cascata, que era a maneira como a maioria das pessoas pensava sobre projetos de software em 1975, infelizmente foi consagrado no DOD-STD-2167, a especificação do Departamento de Defesa para todos os softwares militares. Isso garantiu sua sobrevivência bem depois do tempo em que a maioria dos praticantes atenciosos reconheceram sua inadequação e a abandonaram. Felizmente, o DoD começou a ver a luz.⁹

Tem que haver movimento rio acima. Como o salmão enérgico na imagem de abertura do capítulo, a experiência e as ideias de cada parte a jusante do processo de construção devem saltar rio acima, às vezes mais de um estágio, e afetar o

atividade upstream.

Projetar a implementação mostrará que alguns recursos arquitetônicos prejudicam o desempenho; então a arquitetura tem que ser retrabalhada. Codificar a realização mostrará algumas funções para os requisitos de espaço do balão; portanto, pode ser necessário fazer alterações na arquitetura e na implementação.

Pode-se muito bem, portanto, iterar por dois ou mais ciclos de design de implementação de arquitetura antes de realizar qualquer coisa como código.

Um modelo de construção incremental é melhor - refinamento progressivo

Construindo um sistema esqueleto de ponta a ponta

Harlan Mills, trabalhando em um ambiente de sistema em tempo real, desde cedo defendeu que devemos construir o loop de pesquisa básico de um sistema em tempo real, com chamadas de sub-rotina (*stubs*) para todas as funções (Fig. 19.2), mas apenas sub-rotinas nulas. Compile-o; teste-o. Ele dá voltas e mais voltas, literalmente sem fazer nada, mas fazendo isso corretamente.¹⁰

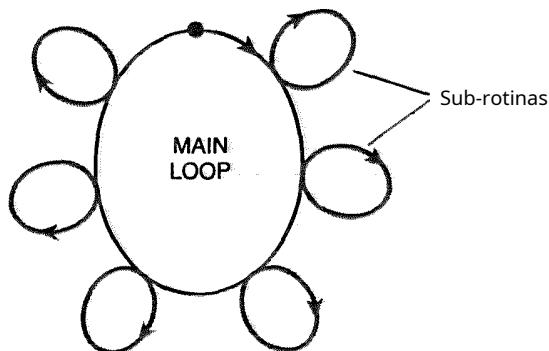


Fig. 19.2

A seguir, elaboramos um módulo de entrada (talvez primitivo) e um módulo de saída. Voilà! Um sistema em execução que faz algo, por mais enfadonho que seja. Agora, função por função, construímos e adicionamos módulos de forma incremental. *Em cada etapa, temos um sistema em execução.* Se formos diligentes, teremos em cada estágio um sistema testado e depurado. (À medida que o sistema cresce, aumenta também o fardo do teste de regressão de cada novo módulo em relação a todos os casos de teste anteriores.)

Depois que cada função funciona em um nível primitivo, refinamos ou reescrevemos primeiro um módulo e depois outro, de forma incremental crescente o sistema. Às vezes, com certeza, temos que mudar o loop de condução original e ou mesmo suas interfaces de módulo.

Uma vez que temos um sistema funcionando em todos os momentos

- podemos começar o teste do usuário muito cedo, e
- podemos adotar uma estratégia de construção de acordo com o orçamento que protege totalmente contra estouros de cronograma ou orçamento (às custas de possíveis diminuições funcionais).

Por cerca de 22 anos, ensinei o laboratório de engenharia de software da Universidade da Carolina do Norte, às vezes junto com David Parnas. Neste curso, equipes de geralmente quatro alunos construíram em um semestre algum sistema de aplicação de software real. Na metade desses anos, passei a ensinar desenvolvimento incremental. Fiquei surpreso com o efeito eletrizante sobre o moral da equipe daquela primeira imagem na tela, aquele primeiro sistema em execução.

ParnasFamilies

David Parnas foi um grande líder de pensamento em engenharia de software durante todo esse período de 20 anos. Todos estão familiarizados com seu conceito de ocultação de informações. Um pouco menos familiar, mas muito importante, é o conceito de Parnas de projetar um produto de software como um *Família* de produtos relacionados. Ele incentiva o designer a prever extensões laterais e versões sucessivas de

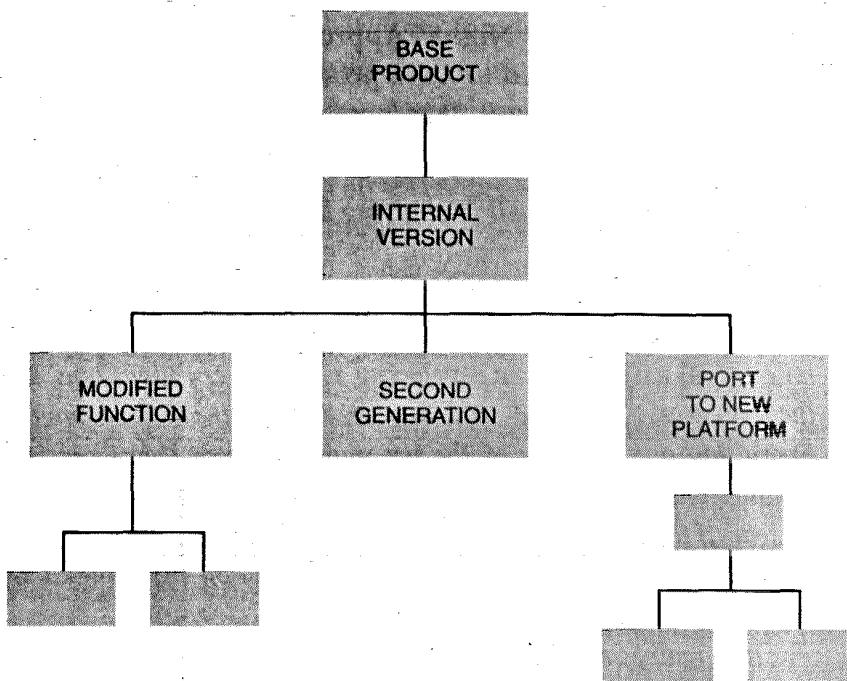


Fig. 19.3

um produto, e para definir sua função ou diferenças de plataforma de modo a construir uma árvore genealógica de produtos relacionados (Fig. 19.3).

O truque no projeto de tal árvore é colocar perto de sua raiz as decisões de projeto que são menos prováveis de serem alteradas.

Essa estratégia de design maximiza a reutilização de módulos. Mais importante, a mesma estratégia pode ser ampliada para incluir não apenas produtos entregáveis, mas também as versões intermediárias sucessivas criadas em uma estratégia de construção incremental. O produto então cresce através de seus estágios intermediários com retrocesso mínimo.

"Build Every" da Microsoft Noite " Abordagem

James McCarthy descreveu para mim um processo de produto usado por sua equipe e outros na Microsoft. É um crescimento incremental levado a uma conclusão lógica. Ele diz,

Depois de enviarmos pela primeira vez, estaremos enviando versões posteriores que adicionam mais funções a um produto existente em execução. Por que o processo de construção inicial deveria ser diferente? Começando no momento de nosso primeiro marco [onde a marcha para o primeiro navio tem três marcos intermediários], reconstruímos o sistema de desenvolvimento todas as noites [e executamos os casos de teste]. O ciclo de construção se torna a pulsação do projeto. Todos os dias, uma ou mais equipes de programadores-testadores fazem check-in de módulos com novas funções. Após cada construção, temos um sistema em execução. Se a compilação for interrompida, paramos todo o processo até que o problema seja encontrado e corrigido. Em todos os momentos, todos na equipe conhecem seu status.

É muito difícil. É preciso dedicar muitos recursos, mas é um processo disciplinado, controlado e conhecido. Isso dá credibilidade à equipe para si mesma. Sua credibilidade determina sua moral, seu estado emocional.

Os desenvolvedores de software em outras organizações ficam surpresos, até mesmo chocados, com esse processo. Um diz: "Tenho o hábito de construir todas as semanas, mas acho que seria muito trabalhoso construir todas as noites." E isso pode ser verdade. A Bell Northern Research, por exemplo, reconstrói seu sistema de 12 milhões de linhas todas as semanas.

Incremental-BuildandRapidPrototyping

Uma vez que um processo de desenvolvimento incremental permite testes iniciais com usuários reais, qual é a diferença entre isso e a prototipagem rápida? Parece-me que os dois estão relacionados, mas separados. Um pode ter um sem o outro.

Harel define utilmente um protótipo como

[Uma versão de um programa que] reflete apenas as decisões de design feitas no processo de preparação do modelo conceitual, e não decisões orientadas por questões de implementação.¹²

É possível construir um protótipo que não faça parte de um produto que está crescendo em direção ao embarque. Por exemplo, pode-se construir um protótipo de interface que não tenha nenhuma função de programa real por trás dele, apenas a máquina de estado finito que faz com que pareça estar em funcionamento. Pode-se até prototipar e testar interfaces pela técnica do Mágico de Oz, com um ser humano oculto simulando as respostas do sistema. Tal prototipagem pode ser muito útil para obter feedback inicial do usuário, mas é totalmente diferente de testar um produto que está crescendo até a remessa.

Da mesma forma, os implementadores podem comprometer-se a construir uma fatia vertical de um produto, na qual um conjunto de funções muito limitado é totalmente construído, de modo a permitir que a luz do sol precoce entre em locais onde as cobras de desempenho podem se esconder. Qual é a diferença entre a construção do primeiro marco do processo da Microsoft e um protótipo rápido? Função. O produto do primeiro marco pode não ter função suficiente para ser do interesse de ninguém; o produto expedível é definido como tal por sua completude no fornecimento de um conjunto útil de funções e, por sua qualidade, a crença de que funciona de maneira robusta.

Parnas estava certo e eu estava errado sobre o ocultamento de informações

No Capítulo 7, contraste duas abordagens para a questão de quanto cada membro da equipe deve ser permitido ou encorajado a saber sobre os designs e o código de cada um. No projeto Sistema Operacional / 360, decidimos que /á os programadores deveriam ver /á o material - isto é, cada programador tendo uma cópia da pasta de trabalho do projeto, que chegou a somar mais de 10.000 páginas. Harlan Mills argumentou de forma persuasiva que "a programação deve ser um processo público", que expor todo o trabalho ao olhar de todos ajuda no controle de qualidade, tanto pela pressão dos colegas para fazer as coisas bem quanto pelos colegas realmente identificando falhas e bugs.

Esta visão contrasta fortemente com o ensino de David Parnas de que os módulos de código devem ser encapsulados com interfaces bem definidas, e que o interior de tal módulo deve ser o

propriedade privada de seu programador, não perceptível de fora. Os programadores são mais eficazes se protegidos, não expostos às entradas de módulos que não são seus.¹³

Rejeitei o conceito de Parnas como uma "receita para o desastre" no Capítulo 7. Parnas estava certo e eu errado. Agora estou convencido de que a ocultação de informações, hoje muitas vezes incorporada na programação orientada a objetos, é a única maneira de elevar o nível do design de software.

De fato, pode-se obter desastres com qualquer uma das técnicas. A técnica de Mills garante que os programadores possam saber a semântica detalhada das interfaces com as quais trabalham, sabendo o que há do outro lado. Ocultar essa semântica leva a bugs do sistema. Por outro lado, a técnica de Parnas é robusta sob mudanças e é mais apropriada em uma filosofia de design para mudanças.

O Capítulo 16 argumenta o seguinte:

- A maior parte do progresso anterior em produtividade de software veio da eliminação de dificuldades não inherentes, como linguagens de máquina complicadas e lentidão no processamento do lote.
- Não há muito mais dessas escolhas fáceis.
- O progresso radical terá que vir do ataque às dificuldades essenciais de moldar construções conceituais complexas.

A maneira mais óbvia de fazer isso reconhece que os programas são compostos de blocos conceituais muito maiores do que a instrução individual da linguagem de alto nível - sub-rotinas, módulos ou classes. Se pudermos limitar o design e a construção de modo que apenas façamos a montagem e parametrização de tais pedaços de coleções pré-construídas, elevamos radicalmente o nível conceitual e eliminamos a vasta quantidade de trabalho e as abundantes oportunidades de erro que habitam o indivíduo nível de declaração.

A definição de módulos que ocultam informações de Parnas é a primeira etapa publicada nesse programa de pesquisa crucialmente importante, e é um ancestral intelectual do programa orientado a objetos.

gramming. Eu defini um módulo como uma entidade de software com seu próprio modelo de dados e seu próprio conjunto de operações. Seus dados só podem ser acessados por meio de uma de suas operações adequadas. A segunda etapa foi uma contribuição de vários pensadores: a atualização do módulo Parnas em um *tipo de dados abstratos*, a partir do qual muitos objetos podem ser derivados. O tipo de dado abstrato fornece uma maneira uniforme de pensar e especificar interfaces de módulo e uma disciplina de acesso que é fácil de aplicar.

A terceira etapa, programação orientada a objetos, apresenta o poderoso conceito de *herança*, em que as classes (tipos de dados) assumem como padrão atributos especificados de seus ancestrais na hierarquia de classes.¹⁴ Muito do que esperamos ganhar com a programação orientada a objetos deriva de fato da primeira etapa, encapsulamento de módulo, além da ideia de bibliotecas de módulos pré-construídas ou aulas *que são projetados e testados para reutilização*. Muitas pessoas têm escolhido para ignorar o fato de que tais módulos não são apenas programas, mas em vez disso são produtos de programa no sentido discutido no Capítulo 1. Algumas pessoas estão esperando em vão por uma reutilização significativa de módulo sem pagar o custo inicial de construção de módulos de qualidade de produto - generalizado, robusto, testado e documentado. A programação orientada a objetos e a reutilização são discutidas nos Capítulos 16 e 17.

Quão mítico é o homem-mês? Modelo e dados de Boehm

Ao longo dos anos, houve muitos estudos quantitativos sobre a produtividade do software e os fatores que a afetam, especialmente as compensações entre a equipe do projeto e o cronograma.

O estudo mais substancial é aquele feito por Barry Boehm de cerca de 63 projetos de software, principalmente aeroespacial, com cerca de 25 em TRW. Seu *Economia da Engenharia de Software* contém não apenas os resultados, mas um conjunto útil de modelos de custo de abrangência progressiva. Considerando que os coeficientes nos modelos são certamente diferentes para software comercial comum e para software aeroespacial construído de acordo com os padrões do governo, no entanto, seu modelo

els são apoiados por uma imensa quantidade de dados. Acho que o livro será um clássico útil daqui a uma geração.

Seus resultados confirmam solidamente a afirmação de MM-M de que o trade-off entre homens e meses está longe de ser linear, que o homem-mês é de fato mítico como medida de produtividade. Em particular, encontrei:¹⁵

- Existe um tempo de programação de custo otimizado para a primeira remessa, $t = 2,5 (\text{MILÍMETROS})^{1/3}$. Ou seja, o tempo ótimo em meses vai como a raiz cúbica do esforço esperado em homens-mês, um valor derivado da estimativa de tamanho e outros fatores em seu modelo. Uma curva ótima de pessoal é um corolário.

¹⁶ A curva de custo sobe lentamente conforme o cronograma planejado começa mais do que o ideal. Pessoas com mais tempo levam mais tempo.

¹⁷ A curva de custo aumenta drasticamente à medida que o cronograma planejado começa mais curto do que o ideal.

- *Quase nenhum projeto é bem-sucedido em menos de 3/4 do cronograma ideal calculado, independentemente do número de pessoas inscritas!* Este resultado citável dá ao gerente de software munição sólida quando a alta gerência está exigindo compromissos impossíveis de cronograma.

Quão verdadeira é a Lei de Brooks? Houve até estudos cuidadosos avaliando a verdade da Lei de Brooks (intencionalmente simplista), de que adicionar mão de obra a um projeto de software atrasado o torna mais tarde. O melhor tratamento é o de Abdel-Hamid e Madnick, em seu ambicioso e valioso livro de 1991, *Software de dinâmica de projeto: uma abordagem integrada*.¹⁸ O livro desenvolve um modelo quantitativo da dinâmica do projeto. Seu capítulo sobre a Lei de Brooks fornece uma visão mais detalhada sobre o que acontece sob várias suposições quanto a que mão de obra é adicionada e quando. Para investigar isso, os autores estendem seu próprio modelo cuidadoso de um projeto de aplicativos de médio porte, assumindo que novas pessoas têm uma curva de aprendizado e levando em consideração a comunicação extra.

trabalho de comunicação e treinamento. Eles concluem que "Adicionar mais pessoas a um projeto atrasado sempre o torna mais caro, mas não *sempre* fazer com que seja concluído mais tarde [itálicos deles]. "Em particular, adicionar mão de obra extra no início do cronograma é uma manobra muito mais segura do que adicionará-la mais tarde, uma vez que as novas pessoas também maneiras têm um efeito negativo imediato, que leva semanas para compensar.

Stutzke desenvolve um modelo mais simples para realizar uma investigação semelhante, com um resultado semelhante.¹⁷ Ele desenvolve uma análise detalhada do processo e dos custos de assimilação dos novos trabalhadores, incluindo explicitamente o desvio de seus mentores da tarefa do projeto em si. Ele testa seu modelo em relação a um projeto real no qual a mão de obra foi duplicada com sucesso e o cronograma original alcançado, após um lapso de meio do projeto. Ele trata de alternativas para adicionar mais programadores, especialmente horas extras. Mais valiosos são seus muitos itens de conselhos práticos sobre como novos trabalhadores devem ser adicionados, treinados, apoiados com ferramentas, etc., de modo a minimizar os efeitos prejudiciais de adicioná-los. Especialmente digno de nota é seu comentário de que as novas pessoas adicionadas posteriormente em um projeto de desenvolvimento devem ser integrantes da equipe dispostos a contribuir e trabalhar dentro do processo, e não tentar alterar ou melhorar o próprio processo!

Stutzke acredita que a carga adicional de comunicação em um projeto maior é um efeito de segunda ordem e não o modela. Não está claro se e como Abdel-Hamid e Madnick o levam em consideração. Nenhum dos modelos leva em conta o fato de que a obra deve ser reparticionada, um processo que freqüentemente considero não trivial.

A declaração "absurdamente simplificada" da Lei de Brooks torna-se mais útil por causa desses cuidadosos tratamentos das qualificações adequadas. Em suma, defendo a afirmação direta como a melhor aproximação zero da verdade, uma regra prática para alertar os gerentes contra fazer cegamente a correção instintiva de um projeto atrasado.

Pessoas são tudo (bem, quase tudo)

Alguns leitores acharam curioso que o MM-M dedica a maioria dos ensaios aos aspectos gerenciais da engenharia de software, ao invés de muitas questões técnicas. Esse preconceito era devido em parte à natureza de minha função no IBM Operating System / 360 (agoraMVS / 370). Mais fundamentalmente, surgiu da convicção de que a qualidade das pessoas em um projeto, e sua organização e gerenciamento, são fatores muito mais importantes para o sucesso do que as ferramentas que usam ou as abordagens técnicas que adotam.

Pesquisas subsequentes apoiaram essa convicção. O modelo COCOMO de Boehm descobre que a qualidade da equipe é, de longe, o maior fator de seu sucesso, na verdade, quatro vezes mais potente do que o segundo maior fator. A maior parte da pesquisa acadêmica em engenharia de software se concentrou em ferramentas. Eu admiro e cobiço ferramentas afiadas. No entanto, é encorajador ver os esforços de pesquisa em andamento sobre o cuidado, crescimento e alimentação das pessoas e sobre a dinâmica do gerenciamento de software.

Peopleware. Uma contribuição importante durante os últimos anos foi o livro de DeMarco e Lister de 1987, *Peopleware: Projetos e equipes produtivas*. Sua tese subjacente é que "Os principais problemas do nosso trabalho não são tanto *tecnológico* ás *sociológico* na natureza. "Ele está repleto de joias como:" A função do gerente não é fazer as pessoas trabalharem, é possibilitar que as pessoas trabalhem. "Ele lida com tópicos mundanos como espaço, móveis, refeições em equipe. DeMarco e Lister fornece dados reais de seus Coding War Games que mostram uma correlação impressionante entre o desempenho dos programadores da mesma organização e entre as características do local de trabalho e os níveis de produtividade e defeito.

*O espaço para os melhores desempenhos é mais silencioso, mais privado, melhor protegido contra interrupções e em maior quantidade. . . Isso realmente importa para você. . . se o silêncio, o espaço e a privacidade ajudam seu pessoal atual a fazer um trabalho melhor ou [alternativamente] ajudam você a atrair e manter pessoas melhores?*¹⁹

Recomendo vivamente o livro a todos os meus leitores.

Projetos em movimento. DeMarco e Lister dão atenção considerável à equipe *fusão*, uma propriedade intangível, mas vital. Acho que é o fato de a administração negligenciar a fusão que explica a prontidão que observei em empresas multilocalizadas para mover um projeto de um laboratório para outro.

Minha experiência e observação são limitadas a talvez meia dúzia de movimentos. Nunca vi um de sucesso. Pode-se mover *missões* com sucesso. Mas em todos os casos de tentativas de movimentação de projetos, a nova equipe na verdade recomeçava, apesar de ter uma boa documentação, alguns projetos bem avançados e algumas pessoas da equipe de envio. Acho que é o rompimento da fusão da velha equipe que aborta o produto embrionário, e faz recomeçar.

O poder de abrir mão do poder

Se alguém acredita, como argumentei em muitos lugares neste livro, que a criatividade vem de indivíduos e não de estruturas ou processos, então uma questão central enfrentada pelo gerente de software é como projetar a estrutura e o processo de modo a melhorar, em vez de inibir, criatividade e iniciativa. Felizmente, esse problema não é peculiar às organizações de software, e grandes pensadores trabalharam nele. EF Schumacher, em seu clássico, *Pequeno é bonito: economia como se as pessoas importassem*, propõe uma teoria de organização de empresas para maximizar a criatividade e alegria dos trabalhadores. Para o seu primeiro princípio, ele escolhe o "Princípio da Função Subsidiária" da Encíclica *Quadragesima Anno* do Papa Pio XI:

É uma injustiça e, ao mesmo tempo, um grave mal e uma perturbação da ordem correta atribuir a uma associação cada vez maior o que as organizações inferiores e subordinadas podem fazer. Pois toda atividade social deve por sua própria natureza fornecer ajuda aos membros do corpo social e nunca destruí-los e absorvê-los. . . Os que estão no comando devem ter certeza de que quanto mais perfeitamente uma ordem graduada é

preservada entre as diversas associações, ao observar o princípio da função subsidiária, tanto mais forte será a autoridade e eficácia social e mais feliz e próspera será a condição do Estado. ¹⁹

Schumacher continua a interpretar:

O Princípio da Função Subsidiária nos ensina que o centro ganhará em autoridade e eficácia se a liberdade e a responsabilidade das formações inferiores forem cuidadosamente preservadas, com o resultado de que a organização como um todo será "mais feliz e mais próspera".

*Como essa estrutura pode ser alcançada?... A grande organização será composta por muitas unidades semi-autônomas, que podemos chamar **quase-firmas**. Cada um deles terá uma grande quantidade de liberdade, para dar a maior chance possível de criatividade e **empreendedorismo**. . . . Cada quase-empresa deve ter uma conta de lucros e perdas e um balanço patrimonial.* ^{vinte}

Entre os desenvolvimentos mais interessantes na engenharia de software estão os estágios iniciais de colocar essas idéias organizacionais em prática. Primeiro, a revolução do microcomputador criou uma nova indústria de software de centenas de empresas iniciantes, todas elas começando pequenas e marcadas pelo entusiasmo, liberdade e criatividade. O setor está mudando agora, pois muitas pequenas empresas estão sendo adquiridas por empresas maiores. Resta saber se os compradores maiores compreenderão a importância de preservar a criatividade dos pequenos.

Mais notavelmente, a alta gerência em algumas grandes empresas se comprometeu a delegar poder a equipes individuais de projeto de software, fazendo-as abordar as quase-empresas de Schumacher em estrutura e responsabilidade. Eles estão surpresos e encantados com os resultados.

Jim McCarthy of Microsoft descreveu sua experiência na emancipação de suas equipes:

Cada equipe de recursos (30-40 pessoas) possui seu conjunto de recursos, sua programação e até mesmo seu processo de como definir, construir, enviar. O time é feito

até quatro ou cinco especialidades, incluindo construção, teste e redação. A equipe resolve disputas; os patrões não. Não consigo enfatizar o suficiente a importância da capacitação, da equipe ser responsável por seu sucesso.

Earl Wheeler, chefe aposentado do negócio de software da IBM, contou-me sua experiência na realização da delegação descendente de poder, há muito centralizado nos gerentes de divisão da IBM:

O principal impulso [nos últimos anos] foi delegar poder. Foi como mágica! Melhor qualidade, produtividade, moral. Temos equipes pequenas, sem controle central. As equipes são donas do processo, mas precisam ter um. Eles têm muitos processos diferentes. Eles são donos da programação, mas sentem a pressão do mercado. Essa pressão faz com que eles procurem as ferramentas por conta própria.

As conversas com membros individuais da equipe, é claro, mostram uma apreciação do poder e da liberdade delegada e uma estimativa um pouco mais conservadora de quanto controle realmente foi abandonado. No entanto, a delegação alcançada é claramente um passo na direção certa. Produz exatamente os benefícios previstos por Pio XI: o centro ganha autoridade real delegando poder, e a organização como um todo fica mais feliz e próspera.

Qual é a maior nova surpresa? Milhões de computadores

Todo guru de software com quem conversei admite ter sido pego de surpresa pela revolução do microcomputador e seu desdobramento, a indústria de software encolhida. Esta é, sem dúvida, a mudança crucial das duas décadas desde o MM-M, tem muitas implicações para a engenharia de software.

A revolução do microcomputador mudou a forma como todos usam os computadores. Schumacher declarou o desafio há mais de 20 anos:

O que realmente exigimos dos cientistas e da tecnologia

essência? Eu deveria responder: Precisamos de métodos e equipamentos que sejam baratos o suficiente para que sejam acessíveis a praticamente todos; adequado para aplicação em pequena escala; e

compatível com a necessidade de criatividade do homem. vinte e um

Essas são exatamente as propriedades maravilhosas que a revolução do microcomputador trouxe para a indústria de computadores e seus usuários, agora o público em geral. O americano médio pode agora pagar não apenas por um computador, mas por um pacote de software que, 20 anos atrás, teria custado o salário de um rei. Cada um dos objetivos de Schumacher vale a pena contemplar; vale a pena saborear o grau em que cada um foi alcançado, especialmente o último. Em uma área após outra, novos meios de autoexpressão são acessíveis tanto às pessoas comuns quanto aos profissionais.

Em parte, o aprimoramento vem em outros campos, como aconteceu na criação de software - na remoção de dificuldades accidentais. Manuscritos escritos costumavam ser acidentalmente enrijecidos pelo tempo e custo de redigitação para incorporar mudanças. Em um trabalho de 300 páginas, pode-se ter que redigitar a cada três ou seis meses, mas, no meio, apenas continuamos marcando o manuscrito. Não era possível avaliar facilmente o que as mudanças haviam feito no fluxo da lógica e no ritmo das palavras. Agora, os manuscritos se tornaram incrivelmente fluidos.²²

O computador trouxe fluidez semelhante a muitas outras mídias: desenhos artísticos, planos de construção, desenhos mecânicos, composições musicais, fotografias, sequências de vídeo, apresentações de slides, obras multimídia e até planilhas. Em cada caso, o método manual de produção exigia a recopiação das peças não modificadas volumosas para ver as mudanças no contexto. Agora, desfrutamos para cada meio os mesmos benefícios que o compartilhamento de tempo trouxe para a criação de software - a capacidade de revisar e avaliar instantaneamente o efeito sem perder a linha de pensamento.

A criatividade também é aprimorada por ferramentas auxiliares novas e flexíveis. Para a produção de prosa, por exemplo, agora somos servidos por corretores ortográficos, corretores gramaticais, consultores de estilo, bibliotecários

sistemas gráficos e a capacidade notável de ver páginas formatadas simultaneamente no layout final. Ainda não avaliamos o que as encyclopédias instantâneas ou os recursos infinitos da World Wide Web significarão para a pesquisa improvisada de um escritor.

Mais importante, a nova fluidez da mídia facilita a exploração de muitas alternativas radicalmente diferentes quando um trabalho criativo está apenas tomando forma. Aqui está outro caso em que uma ordem de magnitude em um parâmetro quantitativo, aqui o tempo de mudança, faz uma diferença qualitativa em como alguém executa uma tarefa.

As ferramentas de desenho permitem que os projetistas de edifícios explorem muito mais opções por hora de investimento criativo. A conexão de computadores a sintetizadores, com software para geração ou execução automática de partituras, torna muito mais fácil capturar rabiscos de teclado. A manipulação de fotografia digital, como com o Adobe Photoshop, permite experimentos de minutos que levariam horas em uma câmara escura. As planilhas permitem a fácil exploração de dezenas de cenários alternativos do tipo "e se".

Finalmente, mídias criativas totalmente novas foram possibilitadas pela onipresença do computador pessoal. Os hipertextos, propostos por Vannevar Bush em 1945, são práticos apenas com computadores.^{2,3}

Apresentações e experiências multimídia eram grandes negócios - muitos problemas - antes do computador pessoal e do software rico e barato disponível para ele. Os sistemas de ambiente virtual, ainda não baratos ou onipresentes, se tornarão e serão mais um meio criativo.

A revolução do microcomputador mudou como todos constrói software. Os próprios processos de software da década de 1970 foram alterados pela revolução do microprocessador e pelos avanços tecnológicos que o permitiram. Muitas das dificuldades acidentais desses processos de construção de software foram eliminadas. Computadores individuais rápidos são agora as ferramentas de rotina do desenvolvedor de software, de modo que o tempo de resposta é um conceito quase obsoleto. O computador pessoal de hoje não é apenas mais rápido

do que o supercomputador do I960, é mais rápido do que a estação de trabalho Unix de 1985. Tudo isso significa que a compilação é rápida mesmo nas máquinas mais humildes, e as memórias grandes eliminaram as esperas de vinculação baseada em disco. As memórias grandes também tornam razoável manter as tabelas de símbolos na memória com o código do objeto, portanto, a depuração de alto nível sem recompilação é rotina.

Nos últimos 20 anos, passamos quase totalmente pelo uso do tempo compartilhado como metodologia para a construção de software. Em 1975, o compartilhamento de tempo tinha acabado de substituir a computação em lote como a técnica mais comum. A rede foi usada para dar ao criador de software acesso a arquivos compartilhados e a um poderoso mecanismo compartilhado para compilação, vinculação e teste. Hoje, a estação de trabalho pessoal fornece o mecanismo de computação e a rede principalmente fornece acesso compartilhado aos arquivos que são o produto de trabalho de desenvolvimento da equipe. Os sistemas cliente-servidor tornam o acesso compartilhado para check-in, construção e aplicação de casos de teste um processo diferente e mais simples.

Avanços semelhantes ocorreram em interfaces de usuário. A interface WIMP fornece uma edição muito mais conveniente de textos de programas, bem como de textos em inglês. A tela de 24 linhas e 72 colunas foi substituída por telas de página inteira ou até mesmo de duas páginas, para que os programadores possam ver muito mais contexto para as mudanças que estão fazendo.

Toda a nova indústria de software - Software Shrink-Wrapped

Junto com a indústria de software clássica, explodiu outra. As vendas de unidades de produtos chegam a centenas de milhares, até milhões. Pacotes completos de software rico podem ser adquiridos por menos do que o custo de um programador-dia com suporte. Os dois setores são diferentes em muitos aspectos e coexistem.

A indústria de software clássica. Em 1975, a indústria de software tinha vários componentes identificáveis e um tanto diferentes, todos os quais ainda existem hoje:

«Fornecedores de computadores, que fornecem sistemas operacionais, com

empilhadeiras e utilitários para seus produtos.

- Usuários de aplicativos, como lojas MIS de concessionárias de serviços públicos, bancos, seguradoras e agências governamentais, que criam pacotes de aplicativos para seu próprio uso.
«Construtores de aplicativos personalizados, que contratam para construir propriedades pacotes diversos para usuários. Muitos desses contratados se especializam em aplicações de defesa, onde os requisitos, padrões e procedimentos de marketing são peculiares.
- Desenvolvedores de pacotes comerciais, que na época desenvolviam principalmente aplicativos de grande porte para mercados especializados, como pacotes de análises estatísticas e sistemas CAD.

Tom DeMarco observa a fragmentação da indústria de software clássica, especialmente o componente de usuário do aplicativo:

O que eu não esperava: o campo se dividiu em nichos. Como você faz algo é muito mais uma função do nicho do que o uso de métodos gerais de análise de sistemas, linguagens gerais e técnicas gerais de teste. Ada foi a última das linguagens de uso geral e se tornou uma linguagem de nicho.

No nicho de aplicação comercial de rotina, as linguagens de quarta geração fizeram contribuições poderosas. Boehm diz: "Os 4GLs mais bem-sucedidos são o resultado de alguém codificar uma parte de um domínio de aplicativo em termos de opções e parâmetros." O mais difundido desses 4GLs são geradores de aplicativos e pacotes de comunicação de banco de dados combinados com linguagens de consulta.

Os mundos do sistema operacional se fundiram. Em 1975, operando sistemas abundantes: cada fornecedor de hardware tinha pelo menos um sistema operacional proprietário por linha de produto; muitos tinham dois. Como as coisas são diferentes hoje! Sistemas abertos são a palavra de ordem, e existem apenas cinco ambientes de sistemas operacionais significativos

mentos nos quais as pessoas comercializam pacotes de aplicativos (em ordem cronológica):

- Os ambientes IBMMVS e VM
 «O ambiente DECVMS
 «O ambiente Unix, de uma forma ou de outra
- O ambiente IBMPC, seja DOS, OS-2 ou Windows O
- ambiente Apple Macintosh.

A indústria embalada. Para o desenvolvedor no

Na indústria embalada em papel termoelétrico, a economia é inteiramente diferente daquela da indústria clássica: o custo de desenvolvimento é dividido por grandes quantidades; os custos de embalagem e marketing são elevados.

Na indústria clássica de desenvolvimento de aplicativos internos, o cronograma e os detalhes da função eram negociáveis, o custo de desenvolvimento pode não ser; no mercado aberto ferozmente competitivo, o cronograma e a função dominam bastante os custos de desenvolvimento.

Como seria de se esperar, a economia totalmente diferente deu origem a culturas de programação bastante diferentes. A indústria clássica tendia a ser dominada por grandes empresas com estilos de gestão e culturas de trabalho estabelecidos. A indústria encolhida, por outro lado, começou como centenas de start-ups, que giravam livremente e se concentravam ferozmente em fazer o trabalho, e não no processo. Nesse clima, sempre houve um reconhecimento muito maior do talento do programador individual, uma consciência implícita de que grandes designs vêm de grandes designers. A cultura de start-up tem a capacidade de recompensar as estrelas de desempenho na proporção de suas contribuições; na indústria de software clássica, a sociologia das corporações e seus planos de gerenciamento de salários sempre dificultaram isso.

Comprar e Construir - Pacotes embalados por contracção como componentes

Robustez e produtividade de software radicalmente melhores devem ser

tinha apenas subindo um nível e fazendo programas pela composição de módulos ou objetos. Uma tendência especialmente promissora é o uso de pacotes para o mercado de massa como plataformas nas quais produtos mais ricos e customizados são construídos. Um sistema de rastreamento de caminhões é construído em um banco de dados compactado e um pacote de comunicações; o mesmo acontece com um sistema de informação do aluno. Os anúncios de procura em revistas de informática oferecem centenas de pilhas HyperCard e modelos personalizados para Excel, dezenas de funções especiais em Pascal para MiniCad ou funções em AutoLisp para AutoCad.

Metaprogramação. A criação de pilhas HyperCard, modelos do Excel ou funções MiniCad às vezes é chamada de *metaprogramação*, a construção de uma nova camada que personaliza a função de um subconjunto de usuários de um pacote. O conceito de metaprogramação não é novo, apenas ressurgente e renomeado. No início da década de 1960, os fornecedores de computadores e muitas lojas de grandes sistemas de informações de gerenciamento (MIS) tinham pequenos grupos de especialistas que criavam linguagens de programação de aplicativos inteiros a partir de macros em linguagem assembly. A loja MIS da Eastman Kodak tinha uma linguagem de aplicação doméstica definida no macroassembler IBM 7080. Da mesma forma, com o OS / 360 Queued Telecommunications Access Method da IBM, pode-se ler muitas páginas de um programa de telecomunicações em linguagem assembly antes de encontrar uma instrução em nível de máquina. Agora, os pedaços oferecidos pelo metaprogramador são muitas vezes maiores do que essas macros.

Isso realmente ataca a essência. Porque o pacote integrado fenômeno não afeta hoje o programador MIS médio, ainda não é muito visível para a disciplina de engenharia de software. No entanto, ele crescerá rapidamente, porque ataca a essência da construção de construções conceituais. O psiquiatra

O pacote empacotado fornece um grande módulo de função, com uma interface elaborada, mas adequada, e sua estrutura conceitual interna não precisa ser projetada de forma alguma. Produtos de software de alta função, como Excel ou 4th Dimension, são realmente grandes módulos, mas servem como módulos conhecidos, documentados e testados com os quais se constroem sistemas personalizados. Os criadores de aplicativos de próximo nível obtêm riqueza de funções, um tempo de desenvolvimento mais curto, um componente testado, melhor documentação e custo radicalmente mais baixo.

A dificuldade, é claro, é que o pacote de software encolhido é projetado como uma entidade autônoma cujas funções e interfaces metaprogramadores não podem mudar. Além disso, e mais seriamente, os construtores de embalagens embaladas aparentemente têm pouco incentivo para tornar seus produtos adequados como módulos em um sistema maior. Acho que essa percepção está errada, de que existe um mercado inexplorado no fornecimento de pacotes projetados para facilitar o uso de metaprogramadores.

Então, o que é necessário? Podemos identificar quatro níveis de usuários de embalagens embaladas a vácuo:

• O usuário no estado em que se encontra, que opera o aplicativo de maneira direta

maneira, o conteúdo com as funções e a interface que os designers forneceram.

- O metaprogramador, que constrói templates ou funções em cima de um único aplicativo, usando a interface fornecida, principalmente para economizar trabalho para o usuário final.

• O escritor de função externa, que codifica manualmente funções adicionadas

em um aplicativo. Esses são essencialmente novos primitivos de linguagem de aplicativo que chamam para separar módulos de código escritos em uma linguagem de uso geral. É necessário ter a capacidade de fazer a interface dessas novas funções com o aplicativo como comandos interceptados, como retornos de chamada ou como funções sobrecarregadas.

• O metaprogramador que usa um, ou especialmente vários,

aplicativos como componentes em um sistema maior. Este é o usuário cujas necessidades são mal atendidas hoje. Este também é o uso

que promete ganhos substanciais de eficácia na construção de novos aplicativos.

Para este último usuário, um aplicativo encolhido precisa de uma interface documentada adicional, a interface de metaprogramação (MPI). Necessita de vários recursos. Primeiro, o metaprograma precisa estar no controle de um conjunto de aplicativos, enquanto normalmente cada aplicativo assume que está no controle. O conjunto deve controlar a interface do usuário, o que normalmente o aplicativo presume que está fazendo. O conjunto deve ser capaz de invocar qualquer função do aplicativo como se sua string de comando tivesse vindo do usuário. Ele deve receber a saída do aplicativo como se fosse a tela, exceto que precisa da saída analisada em unidades lógicas de tipos de dados adequados, em vez da string de texto que teria sido exibida. Alguns aplicativos, como o FoxPro, têm buracos de minhoca que permitem a passagem de uma string de comando, mas a informação que se obtém é acanhada e não analisada. O buraco de minhoca é um *Ad hoc* solução para uma necessidade que exige uma solução geral e projetada.

É poderoso ter uma linguagem de script para controlar as interações entre o conjunto de aplicativos. O Unix primeiro forneceu esse tipo de função, com seus canais e seu formato de arquivo de string ASCII padrão. Hoje, o AppleScript é um bom exemplo.

O Estado e o Futuro da Engenharia de Software

Certa vez, pedi a Jim Ferrell, presidente do Departamento de Engenharia Química da Universidade Estadual da Carolina do Norte, que relatassem a história da engenharia química, distinta da química. Ele então fez um maravilhoso relato improvisado de uma hora, começando com a existência, desde a antiguidade, de muitos processos de produção diferentes para muitos produtos, do aço ao pão e ao perfume. Ele contou como o Professor Arthur D. Little fundou um Departamento de Química Industrial no MIT em 1918, para encontrar, desenvolver e ensinar uma base comum de técnica compartilhada

por todos os processos. Primeiro vieram as regras práticas, depois os nomogramas empíricos, depois as fórmulas para projetar componentes particulares, depois os modelos matemáticos para o transporte de calor, transporte de massa, transporte de momento em vasos individuais.

À medida que a história de Ferrell se desenrolava, fiquei impressionado com os muitos paralelos entre o desenvolvimento da engenharia química e o da engenharia de software, quase exatamente cinquenta anos depois. Parnas me reprova por escrever sobre *Engenharia de software* em absoluto. Ele compara a disciplina de software com a engenharia elétrica e sente que é uma presunção chamar o que fazemos de engenharia. Ele pode estar certo de que o campo nunca se desenvolverá em uma disciplina de engenharia com uma base matemática tão precisa e abrangente quanto a da engenharia elétrica. Afinal, a engenharia de software, como a engenharia química, está preocupada com os problemas não lineares de escalonamento em processos em escala industrial e, como a engenharia industrial, é permanentemente confundida pelas complexidades do comportamento humano.

No entanto, o curso e o momento do desenvolvimento da engenharia química me levam a acreditar que a engenharia de software aos 27 anos pode não ser desesperadora, mas meramente imatura, como a engenharia química era em 1945. Foi somente após a Segunda Guerra Mundial que os engenheiros químicos realmente abordaram o comportamento do circuito fechado sistemas de fluxo contínuo interconectados.

As preocupações distintas da engenharia de software são hoje exatamente aquelas estabelecidas no Capítulo 1:

- Como projetar e construir um conjunto de programas em um *sistema*
- “Como projetar e construir um programa ou sistema em um ro-
- busto, testado, documentado, suportado *produtos*
- Como manter o controle intelectual sobre *complexidade* em grandes doses.

O alcatrão da engenharia de software continuará a ser difícil por muito tempo. Pode-se esperar que a raça humana continue tentando sistemas apenas dentro ou fora de nosso alcance; e os sistemas de software são talvez os mais intrincados do homem

diworks. Este complexo ofício exigirá nosso desenvolvimento contínuo da disciplina, nosso aprendizado para compor em unidades maiores, nosso melhor uso de novas ferramentas, nossa melhor adaptação de métodos comprovados de gestão de engenharia, aplicação liberal do bom senso e uma humildade divina para reconhecer nossa falibilidade e limitações.

Cinquenta anos de maravilha, entusiasmo e alegria

Ainda vívido em minha mente está a maravilha e o deleite com que eu - então com 13 anos - li o relato da dedicação do computador Harvard Mark I em 7 de agosto de 1944, uma maravilha eletromecânica da qual Howard Aiken foi o arquiteto e engenheiros da IBM Clair Lake, Benjamin Durfee e Francis Hamilton foram os designers de implementação. Igualmente instigante foi a leitura do artigo "That We May Think" de Vannevar Bush em abril de 1945 *Atlantic Monthly*, no qual ele propôs organizar o conhecimento como uma grande teia de hipertexto e dar aos usuários máquinas para seguir as trilhas existentes e abrir novas trilhas de associações.

Minha paixão por computadores teve outro forte impulso em 1952, quando um emprego de verão na IBM em Endicott, Nova York, me deu experiência prática na programação do IBM 604 e instrução formal na programação do 701 da IBM, sua primeira máquina de programa armazenado. A pós-graduação de Aiken e Iverson em Harvard tornou meu sonho de carreira uma realidade, e fiquei viciado para o resto da vida. A apenas uma fração da raça humana Deus dá o privilégio de ganhar o pão fazendo o que se teria feito de graça, por paixão. Estou muito agradecido.

É difícil imaginar uma época mais emocionante para ter vivido como um devoto do computador. De mecanismos a tubos de vácuo a transistores a circuitos integrados, a tecnologia explodiu. O primeiro computador em que trabalhei, recém saído de Harvard, foi o

Supercomputador IBM 7030 Stretch. O Stretch reinou como o computador mais rápido do mundo de 1961 a 1964; nove cópias foram entregues. Meu Macintosh Powerbook hoje não é apenas mais rápido, com uma memória e um disco maiores, é mil vezes mais barato. (*Cinco mil vezes mais barato em dólares constantes.*) Vimos, por sua vez, a revolução do computador, a revolução do computador eletrônico, a revolução do minicomputador e a revolução do microcomputador, cada uma trazendo ordens de magnitude mais computadores.

A disciplina intelectual relacionada ao computador explodiu, assim como a tecnologia. Quando eu era um estudante de graduação em meados da década de 1950, podia ler */á* as revistas e anais de conferências; Eu poderia ficar atualizado em */á* a disciplina. Hoje, minha vida intelectual me viu, com pesar, beijando os interesses da subdisciplina, um por um, enquanto meu portfólio continuamente transbordou além do domínio. Muitos interesses, muitas oportunidades interessantes de aprendizagem, pesquisa e reflexão. Que situação maravilhosa! Não só o fim não está à vista, como o ritmo não está diminuindo. Temos muitas alegrias futuras.

Notas e Referências

Capítulo 1

1. Ershov considera isso não apenas uma dor, mas também uma parte da alegria. AP Ershov, "Estética e o fator humano na programação / 'CACM, 15, 7 (julho de 1972), pp. 501-505.

Capítulo 2

1. VA Vyssotsky, da Bell Telephone Laboratories, estima que um grande projeto pode sustentar um aumento de mão de obra de 30% ao ano. Mais do que isso restringe e até inibe a evolução da estrutura informal essencial e suas vias de comunicação discutidas no Capítulo 7.
FJ Corbat <5 do MIT aponta que um projeto longo deve prever uma rotatividade de 20 por cento ao ano, e estes devem ser treinados tecnicamente e integrados à estrutura formal.
2. C. Portman, da International Computers Limited, diz:
"Quando tudo estiver funcionando, tudo integrado, você terá mais quatro meses de trabalho pela frente." Vários outros conjuntos de divisões de cronograma são fornecidos em Wolverton, RW, "O custo de desenvolvimento de software em grande escala", IEEE Trans, em computadores, C-23, 6 (junho de 1974) pp. 615-636.
3. As Figuras 2.5 a 2.8 são devidas a Jerry Ogdin, que ao citar meu exemplo de uma publicação anterior deste capítulo melhorou muito sua ilustração. Ogdin, JL, "The Mongolian hordes versus superprogrammer", Infosystems (Dezembro, 1972), pp. 20-23.

Capítulo 3

1. Sackman, H., WJ Erikson e EE Grant, "Exploratory experimental studies comparing online and offline programming performance," *CACM*, 11, 1 (janeiro de 1968), pp. 3-onze.
2. Mills, H., "Equipes de programador chefe, princípios e procedimentos," IBM Federal Systems Division Report FSC 71-5108, Gaithersburg, Md., 1971.
3. Baker, FT, "Gerenciamento da equipe de programador chefe de programação de produção", *IBM Sys. J.*, 11, 1 (1972).

Capítulo 4

1. Eschapasse, M., *Catedral de Reims*, Caisse Nationale des Monuments Historiques, Paris, 1967.
2. Brooks, FP, "Arquitetura da filosofia", em W. Buchholz (ed.), *Planejando um sistema de computador*. Nova York: McGraw-Hill, 1962.
3. Blaauw, GA, "Requisitos de hardware para a quarta geração", em F. Gruenberger (ed.), *Computadores de quarta geração*. Englewood Cliffs, NJ: Prentice-Hall, 1970.
4. Brooks, FP e KE Iverson, *Processamento automático de dados, System / 360 Edition*. Nova York: Wiley, 1969, Capítulo 5.
5. Glegg, GL, *O Design do Design*. Cambridge: Cambridge Univ. Press, 1969, diz "À primeira vista, a ideia de quaisquer regras ou princípios sendo sobrepostos à mente criativa parece mais provavelmente atrapalhar do que ajudar, mas isso não é verdade na prática. O pensamento disciplinado focaliza a inspiração, em vez de piscá-la. "
- 6 Conway, RW, "The PL / C Compiler," *Proceedings of a Conf. On Definition and Implementation of Universal Programming Languages*. Stuttgart, 1970.
7. Para uma boa discussão sobre a necessidade de tecnologia de programação, consulte CH Reynolds, "O que há de errado com o com-

gerenciamento de programação de computador? "em G. F. Weinwurm (ed.), *Sobre a gestão da programação de computadores*. Filadélfia: Auerbach, 1971, pp. 35-42.

capítulo 5

1. Strachey, C., "Review of *Planning a Computer System*, "Comp. /., 5, 2 (julho de 1962), pp. 152-153.
2. Isso se aplica apenas aos programas de controle. Algumas das equipes de compiladores no esforço do OS / 360 estavam construindo seu terceiro ou quarto sistema, e a excelência de seus produtos mostra isso.
3. Shell, DL, "O sistema Share 709: um esforço cooperativo"; Greenwald, ID e M. Kane, "O sistema Share 709: programação e modificação"; Boehm, EM e TB Steel, Jr., "O sistema Share 709: implementação de máquina de programação simbólica"; all in / ACM, 6, 2 (abril de 1959), pp. 123-140.

Capítulo 6

1. Neustadt, RE, *Poder presidencial*. Nova York: Wiley, 1960, Capítulo 2.
2. Backus, JW, "A sintaxe e semântica da linguagem algébrica internacional proposta." *Proc, Intl. Com/. Inf. Proc. UNESCO*, Paris, 1959, publicado por R. Oldenbourg, Munich e Butterworth, London. Além disso, toda uma coleção de artigos sobre o assunto está contida em TB Steel, Jr. (ed.), *Linguagens de descrição de linguagem formal para programação de computadores*. Amsterdam: North Holland, (1966).
3. Lucas, P. e K. Walk, "Sobre a descrição formal de PL / I," *Revisão Anual em Linguagem de Programação Automática*. New York: Wiley, 1962, Capítulo 2, p. 2
4. Iverson, KE, A *Linguagem de programação*. Nova York: Wiley, 1962, Capítulo 2.

5. Falkoff, AD, KE Iverson, EH Sussenguth, "A formal description of System / 360," *ZBM Systems Journal*, 3, 3 (1964), pp. 198-261.
6. Bell, CG e A. Newell, *Estruturas de computador*. Nova York: McGraw-Hill, 1970, pp. 120-136, 517-541.
7. Bell, CG, comunicação privada.

Capítulo 7

1. Parnas, DL, "Aspectos de distribuição de informação da metodologia de design", Carnegie-Mellon Univ., Dept. of Computer Science Technical Report, fevereiro de 1971.
2. Copyright 1939, 1940 Street & Smith Publications, Copyright 1950, 1967 de Robert A. Heinlein. Publicado por acordo com a Spectrum Literary Agency.

Capítulo 8

1. Sackman, H., WJ Erikson e EE Grant, "Exploratory experimentation studies comparing online and offline programming performance," *CACM*, 11, 1 (Jan., 1968), pp. 3-onze.
2. Nanus, B., and L. Farr, "Alguns contribuintes de custos para programas em grande escala", *AFIPS Proc. SJCC*, 25 (Spring, 1964), pp. 239-248.
3. Weinwurm, GF, "Research in the management of computer programming," Report SP-2059, System Development Corp., Santa Monica, 1965.
4. Morin, LH, "Estimation of resources for computer programming projects," MS thesis, Univ. Of North Carolina, Chapel Hill, 1974.
5. Portman, C., comunicação privada.
6. Um estudo de 1964 não publicado por EF Bardain mostra programadores realizando 27 por cento do tempo produtivo. (Citado por

DB Mayer e A. W. Stalnaker, "Seleção e avaliação do pessoal de informática /' Proc. 23º ACM. Com /., 1968, p. 661.)

7. Aron, J., comunicação privada.
8. Trabalho apresentado em uma sessão de painel e não incluído no *Procedimentos AFIPS*.
9. Wolverton, RW, "O custo de desenvolvimento de software em grande escala," *IEEE Trans, em computadores*, C-23, 6 (junho de 1974) pp. 615-636. Este importante artigo recente contém dados sobre muitas das questões deste capítulo, bem como confirma as conclusões de produtividade.
10. Corbato, FJ, "Sensitive issues in the design of multi-use systems", palestra na abertura do Honeywell EDP TechnologyCenter, 1968.
11. WM Taliaffero também reporta uma produtividade constante de 2.400 extratos / ano em montadora, Fortran e Cobol. Consulte "Modularidade. A chave para o potencial de crescimento do sistema", *Programas, EU*, 3 (julho de 1971) pp. 245-257.
12. SystemDevelopment Corp. ReportTM-3225 da EA Nelson, *Manual de Gestão para a Estimativa de Programa de Computador-custos de míng*, mostra uma melhoria de produtividade de 3 para 1 para linguagem de alto nível (págs. 66-67), embora seus desvios padrão sejam amplos.

Capítulo 9

1. Brooks, FP e KE Iverson, *Processamento automático de dados, System / 360 Edition*. Nova York: Wiley, 1969, Capítulo 6.
2. Knuth, DE, *A Arte da Programação de Computador*, Vols. 1-3. Reading, Mass.: Addison-Wesley, 1968, ss.

Capítulo 10

1. Conway, ME, "Como os comitês inventam?" *Datamation*, 14, 4 (abril de 1968), pp. 28-31.

Capítulo 11

1. Discurso na Oglethorpe University, 22 de maio de 1932.
 2. Um relato esclarecedor da experiência do Multics em dois sistemas sucessivos está em FJ Corbatd, JH Saltzer e CT Clingen, "Multics - os primeiros sete anos", *AFIPS Proc S / CC*, 40 (1972), pp. 571-583.
 3. Cosgrove, J., "Needed: a new planning framework," *Datamation*, 17, 23 (dezembro de 1971), pp. 37-39.
 4. A questão da mudança de design é complexa, e simplifico demais aqui. Veja JH Saltzer, "Evolutionary design of complex systems", em D. Eckman (ed.), *Sistemas: Pesquisa e Design*. NewYork: Wiley, 1961. No entanto, quando tudo estiver dito e feito, ainda defendo a construção de um sistema piloto cujo descarte é planejado.
 5. Campbell, E., "Report to the AEC Computer Information Meeting", dezembro de 1970. O fenômeno também é discutido por JL Ogdin em "Projetando um software confiável", *Datamation*, 18, 7 (julho de 1972), pp. 71-78. Meus amigos experientes parecem divididos de forma bastante equilibrada sobre se a curva finalmente desce novamente.
 6. Lehman, M. e L. Belady, "Programming system dynamics", apresentado no ACM SIGOPS Third Symposium on Operating System Principles, outubro de 1971.
- 7 Lewis, CS, *Eu mereço o cristianismo*. Nova York: Macmillan, I960, p. 54

Capítulo 12

1. Consulte também JW Pomeroy, "Um guia para ferramentas e técnicas de programação", *IBM Sys. J.*, 11, 3 (1972), pp. 234-254.
2. Landy, B. e RM Needham, "Técnicas de engenharia de software usadas no desenvolvimento do Cambridge Multiple-Access System," *Programas*, 1, 2 (abril de 1971), pp. 167-173

3. Corbatd, FJ, "PL / I as a tool for systemprogramming," *Datamation*, quinze, 5 (Maio de 1969), pp. 68-76.
4. Hopkins, M., "Problems of PL / I for system programming," IBM Research Report RC 3489, Yorktown Heights, NY, 5 de agosto de 1971.
5. Corbatd, FJ, JH Saltzer e CT Clingen, "Multics - the first seven years," *AFIPS Proc SJCC*, 40 (1972), pp. 571-582. *"Apenas meia dúzia de áreas que foram escritas em PL / I foram retrocedidas em linguagem de máquina por razões de espremer o máximo em desempenho. Vários programas, originalmente em linguagem de máquina, foram recuados em PL / I para aumentar sua capacidade de manutenção* ity. "
- 6 Para citar o artigo de Corbato citado na referência 3: " *PL / I está aqui agora e as alternativas ainda não foram testadas.* " Mas veja uma visão totalmente contrária, bem documentada, em Henricksen, JO e RE Merwin, "Eficiência da linguagem de programação em sistemas de software em tempo real", *AFIPS Proc SJCC*, 40 (1972) pp. 155-161.
7. Nem todos concordam. Harlan Mills disse, em uma comunicação privada, " *Minha experiência começa a me dizer que na programação da produção quem vai colocar no terminal é a secretária. A ideia é tornar a programação uma prática mais pública, sob o escrutínio comum de muitos membros da equipe, ao invés de uma arte privada.* "
8. Harr, J., "Programming Experience for the Number 1 Electronic Switching System", artigo apresentado no 1969 SJCC.

Capítulo 13

1. Vyssotsky, VA, "Common sense in design testable software", palestra no The Computer Program Test Methods Symposium, Chapel Hill, NC, 1972. A maior parte da palestra de Vyssotsky está contida em Hetzel, WC (ed.), *Métodos de teste do programa*. Englewood Cliffs, NJ: Prentice-Hall, 1972, pp. 41-47
2. Wirth, N., "Program development by stepwise refinement", CACM 14, 4 (abril, 1971), pp. 221-227. Veja também

- Mills, H. "Top-down programming in large systems", em R. Rustin (ed.) - *Técnicas de depuração em grandes sistemas*. Englewood Cliffs, NJ: Prentice-Hall, 1971, pp. 41-55 e Baker, FT, "Qualidade do sistema por meio de programação estruturada," *AFIPS Proc FJCC*, 41-1 (1972), pp. 339-343.
3. Dahl, OJ, EW Dijkstra e CAR Hoare, *Programação Estruturada*. Londres and NewYork: Academic Press, 1972. Este volume contém o tratamento mais completo. Veja também a carta germinal de Dijkstra, "declaração GOTO considerada prejudicial", *CACM*, II, 3 (março de 1968), pp. 147-148.
 4. Bohm, C. e A. Jacopini, "Diagramas de fluxo, máquinas de Turing e linguagens com apenas duas regras de formação", *CACM*, 9, 5 (maio, 1966), pp. 366-371.
 5. Codd, EF, ES Lowry, E. McDonough, e CA Scalzi, "Multiprogramming STRETCH: Feasibility Considerations," *CACM*, 2.11 (Nov., 1959), pp. 13-17.
 6. Strachey, C., "Time sharing in large fast computers," *Proc. Int. Com/ no processamento de informações*, UNESCO (junho de 1959), pp. 336-341. Ver também as observações de Codd na p. 341, onde relatei o progresso de um trabalho como o proposto no artigo de Strachey.
 7. Corbat <5, FJ, M. Merwin-Daggett, RC Daley, "An experimental time-sharing system," *AFIPS Proc. SJCC*, 2, (1962), pp. 335-344. Reimpresso em S. Rosen, *Sistemas de programação e idiomas*. NewYork: McGraw-Hill, 1967, pp. 683-698.
 8. Gold, MM, "Uma metodologia para avaliar o uso de sistema de computador compartilhado por tempo", Ph.D. dissertação, Carnegie-Mellon University, 1967, p. 100
 9. Gruenberger, F., "Teste e validação do programa," *Datamation*, 14, 7, (julho de 1968), pp. 39 ^ 7.
 10. Ralston, A., *Introdução à Programação e Ciência da Computação*. NewYork: McGraw-Hill, 1971, pp. 237-244.
 11. Brooks, FP e KE Iverson, *Processamento automático de dados, System / 360 Edition*. Nova York: Wiley, 1969, pp. 296-299.

12. Um bom tratamento de desenvolvimento de especificações e de construção e teste de sistema é dado por FM Trapnell, "Uma abordagem sistemática para o desenvolvimento de programas de sistema," *AFIPS Proc S / CC*, 34 (1969) pp. 411-418.
13. Um sistema em tempo real exigirá um simulador de ambiente. Veja, por exemplo, MG Ginzberg, "Notas sobre o teste de programas do sistema em tempo real," *IBMSys. /.*, 4, 1 (1965), pp. 58-72.
14. Lehman, M. e L. Belady, "Programming system dynamics", apresentado no ACM SIGOPS Third Symposium on Operating System Principles, outubro de 1971.

Capítulo 14

1. Consulte CH Reynolds, "O que há de errado com o gerenciamento de programação de computador?" em GF Weinwurm (ed.), *On a Gestão de Programação de Computadores*. Filadélfia: Auer-Bach, 1971, pp. 35-42.
2. King, WR e TA Wilson, "Estimativas subjetivas de tempo no planejamento do caminho crítico - uma análise preliminar", *Mgt. Sci.*, 13, 5 (janeiro de 1967), pp. 307-320, e sequela, WR King, D. M. Witterrongel, KD Hezel, "Na análise do comportamento de estimativa do tempo do caminho crítico," *Mgt. Sci.*, 14, 1 (setembro, 1967), pp. 79-S4.
3. Para uma discussão mais completa, consulte Brooks, FP e KE Iverson, *Processamento Automático de Dados, System / 360 Edition*, Nova york: Wiley, 1969, pp. 428-430.
4. Comunicação privada.

Capítulo 15

1. Goldstine, HH e J. von Neumann, "Planning and coding problems for an electronic computing instrument", Parte II, Vol. 1, relatório preparado para o Departamento de Ordenação do Exército dos EUA, 1947; reimpresso em J. von Neumann, *Obras Coletadas*, AH Taub (ed.), Vol. V., New York: McMillan, pp. 80-151.

2. Comunicação privada, 1957. O argumento foi publicado em Iverson, KE, "The Use of APL in Teaching," Yorktown, NY: IBM Corp., 1969.
3. Outra lista de técnicas para PL / I é fornecida por AB Walter e M. Bohl em "Do melhor para o melhor - dicas para uma boa programação," *Idade do software*, 3, 11 (novembro de 1969), pp. 46-50.
As mesmas técnicas podem ser usadas em Algol e até Fortran. DE Lang, da University of Colorado, tem um programa de formatação Fortran denominado ESTILO que realiza esse resultado. Consulte também DD McCracken e GM Weinberg, "How to write a readable FORTRAN program," *Datamation*, 18, 10 (outubro de 1972), pp. 73-77.

Capítulo 16

1. O ensaio intitulado "No Silver Bullet" é de Information Processing 1986, Proceedings of the IFIP Décima Conferência Mundial de Computação, editado por H.-J. Kugler (1986), pp. 1069-76. Reproduzido com a gentil permissão de IFIP e Elsevier Science BV, Amsterdã, Holanda.
2. Parnas, DL, "Projetando software para facilitar a extensão e contração," *IEEE Trans, em SE*, 5.2 (Março, 1979), pp. 128-138
3. Booch, G., "Object-oriented design," in *Engenharia de software com a Ada*. Menlo Park, Califórnia: Benjamin / Cummings, 1983.
4. Mostow, J., ed., Edição Especial sobre Inteligência Artificial e Engenharia de Software, / EEE Trans, em SE, 11, 11 (novembro, 1985).
5. Parnas, DL, "Aspectos de software de sistemas de defesa estratégica," *Comunicações do ACM*, 28, 12 (dez. 1985), pp. 1326-1335. Também em *Cientista americano*, 73, 5 (Setembro-outubro, 1985), pp. 432-440.
6. Balzer, R., "A 15-year perspective on automatic programming," in Mostow, *op. cit.*
7 Mostow, *op. cit.*
8. Parnas, 1985, *op. cit.*

9. Raeder, G., "Um levantamento das técnicas atuais de programação gráfica", em RB Grafton e T. Ichikawa, eds., Edição Especial sobre Programação Visual, *Computador*, 18, 8 (agosto de 1985), pp. 11-25.
10. O tópico é discutido no Capítulo 15 deste livro.
11. Mills, HD, "Top-down programming in large systems," *Técnicas de depuração em grandes sistemas*, R. Rustin, ed., Englewood Cliffs, NJ, Prentice-Hall, 1971.
12. Boehm, BW, "Um modelo espiral de desenvolvimento e aprimoramento de software", *Computador*, 20, 5 (maio, 1985), pp. 43-57.

Capítulo 17

O material citado sem citação provém de comunicações pessoais.

1. Brooks, FP, "Sem bala de prata - essência e acidentes da engenharia de software," in *Processamento de Informações* 86, H. J. Kugler, ed. Amsterdã: Elsevier Science (North Holland), 1986, pp. 1069-1076.
2. Brooks, FP, "Nenhuma bala de prata - essência e acidentes da engenharia de software," *Computador* 20.4 (abril de 1987), pp. 10-19.
3. Várias das cartas e uma resposta apareceram na edição de julho de 1987 de *Computador*.
É um prazer especial observar que, embora "NSB" não tenha recebido nenhum prêmio, a crítica de Bruce M. Skwiersky sobre ela foi selecionada como a melhor crítica publicada em *Críticas de computação* no 1988. EA Weiss, "Editorial", *Críticas de computação* (Junho, 1989), pp. 283-284, ambos anunciam o prêmio e reimprime a crítica de Skwiersky. A revisão contém um erro significativo: "séxtuplo" deveria ser "10⁶."
4. "De acordo com Aristóteles, e na filosofia escolástica, um acidente é uma qualidade que não pertence a uma coisa por direito de sua natureza essencial ou substancial, mas ocorre nela como efeito de outras causas." *Novo Dicionário Internacional da Língua Inglesa Webster*, 2d ed., Springfield, Mass.: GC Merriam, 1960.

5. Sayers, Dorothy L., *A mente do criador*. Nova York: Harcourt, Brace, 1941.
6. Glass, RL e SA Conger, "Research software tasks: Intellectual or clerical?" *Informação e Gestão*, 23, 4 (1992). Os autores relatam que uma medição da especificação de requisitos de software é cerca de 80% intelectual e 20% administrativa. Fjelstadt e Hamlen, 1979, obtém essencialmente os mesmos resultados para manutenção de software de aplicativo. Não conheço nenhuma tentativa de medir essa fração para toda a tarefa de ponta a ponta.
7. Herzberg, F., B. Mausner e BB Sayderman. *A motivação para trabalhar*, 2^a ed. Londres: Wiley, 1959.
8. Cox, BJ, "Há uma bala de prata", *Byte* (Outubro de 1990), pp. 209-218.
9. Harel, D., "Mordendo a bala de prata: Rumo a um futuro mais brilhante para o desenvolvimento de sistemas," *Computador* (Janeiro, 1992), pp. 8-20.
10. Parnas, DL, "Aspectos de software de sistemas de defesa estratégica", *Comunicações do ACM*, 28, 12 (dez. 1985), pp. 1326-1335.
11. Turski, WM, "E nenhuma pedra filosofal, também", em *Processamento de Informações 86*, HJ Kugler, ed. Amsterdam: Elsevier Science (North Holland), 1986, pp. 1077-1080.
12. Glass, RL e SA Conger, "Research Software Tasks: Intellectual or Clerical?" *Informação e Gestão*, 2. 3. 4 (1992), pp. 183-192.
- 13 *Revisão de Computadores Digitais Eletrônicos, Procedimentos de uma Conferência Conjunta de Computadores AIEE-IRE* (Filadélfia, 10 a 12 de dezembro, 1951). NewYork: American Institute of Electrical Engineers, pp. 13-20.
- 14 *Ibid.*, pp. 36, 68, 71, 97.
quinze. *Proceedings of the Eastern Joint Computer Conference*, (Washington, 8 a 10 de dezembro de 1953). Nova York: Institute of Electrical Engineers, pp. 45-47.
- 16 *Anais da Western Joint Computer Conference de 1955* (o

- Angeles, 1-3 de março de 1955). NewYork: Institute of Electrical Engineers.
17. Everett, RR, CA Zraket e HD Bennington, "SAGE - A data processing system for air defence", *Proceedings of the Eastern Joint Computer Conference*, (Washington, 11 a 13 de dezembro de 1957). Nova 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 complex reactive systems," *IEEE Trans, em SE*, 16, 4 (1990), pp. 403 ^ 44.
 19. Jones, C., *Avaliação e controle de riscos de software*. Englewood Cliffs, NJ: Prentice-Hall, 1994. p. 619.
 20. Coqui, H., "Corporate survival: The software dimension," *Focus '89*, Cannes, 1989.
 21. Coggins, JamesM., "Projetando bibliotecas C ++," *C ++ Diário*, 1, 1 (junho de 1990), pp. 25-32.
 22. O tempo verbal é futuro; Não conheço nenhum resultado relatado para um quinto uso.
 23. Jones, op. cit., p 604.
 24. Huang, Weigiao, "Industrializando a produção de software", *Proceedings ACM 1988 Computer Science Conference*, Atlanta, 1988. Temo a falta de crescimento pessoal do emprego em tal arranjo.
 25. Toda a edição de setembro de 1994 do *IEEE Programas* está em reutilização.
 26. Jones, op. cit., p. 323.
 27. Jones, op. cit., p. 329.
 28. Yourdon, E., *Declínio e queda do programador americano*. Englewood Cliffs, NJ: Yourdon Press, 1992, p. 221.
 29. Glass, RL, "Glass" (coluna), *Desenvolvimento de sistema*, (Janeiro, 1988), pp. 4-5.

Capítulo 18

1. Boehm, BW, *Economia da Engenharia de Software*, Englewood Cliffs, NJ: Prentice-Hall, 1981, pp. 81 ^ 84.
2. McCarthy, J., "21 Rules for Delivering Great Software on Time", Software World USA Conference, Washington (setembro, 1994).

Capítulo 19

O material citado sem citação provém de comunicações pessoais.

1. Sobre este doloroso assunto, veja também Niklaus Wirth "A plea for lean software", *Computador*, 28, 2 (fevereiro de 1995), pp. 64-68.
2. Coleman, D., 1994, "Word 6.0 packs in features; update slowed baggage," *MacWeek*, 8, 38 (26 de setembro de 1994), p. 1
3. Muitas pesquisas de linguagem de máquina e frequências de comando de linguagem de programação *depois* de fielding foram publicados. Por exemplo, consulte J. Hennessy e D. Patterson, *Arquitetura do computador*. Esses dados de frequência são muito úteis para construir produtos sucessores, embora nunca se apliquem exatamente. Não conheço estimativas de frequência publicadas preparadas *antes* o produto foi projetado, muito menos comparações de *a priori* estimativas e *a posteriori* dados. Ken Brooks sugere que os quadros de avisos na Internet agora fornecem um método barato de solicitar dados de usuários em potencial de um novo produto, embora apenas um conjunto auto-selecionado responda.
4. Conklin, J. e M. Begeman, "gIBIS: A Hypertext Tool for Exploratory Policy Discussion", *Transações ACM em sistemas de informação de escritório*, Outubro de 1988, pp. 303-331.
5. Englebart, D. e W. English, "A research center for augmenting human intellect," *AFIPS Conference Proceedings, Fall Joint Computer Conference*, San Francisco (9-11 de dezembro de 1968), pp. 395-410.
6. Apple Computer, Inc., *Diretrizes de interface humana do Macintosh*, Reading, Mass.: Addison-Wesley, 1992.

7. Parece que o Apple Desk Top Bus pode lidar com dois mouses eletronicamente, mas o sistema operacional não fornece essa função.
8. Royce, WW, 1970. "Gerenciando o desenvolvimento de grandes sistemas de software: conceitos e técnicas," *Processos*, WESCON (agosto de 1970), reimpresso no *Procedimentos ICSE 9*. Nem Royce nem outros acreditaram que alguém pudesse passar pelo processo de software sem revisar documentos anteriores; o modelo foi apresentado como um ideal e um auxílio conceitual. Ver DL Parnas e PC Clements, "Um processo de design racional: como e por que fingir," *IEEE Transactions on Software Engineering*, SE-12, 2 (fevereiro, 1986), pp. 251-257.
9. Uma grande reformulação do DOD-STD-2167 produziu o DOD-STD2167A (1988), que permite, mas não exige, modelos mais recentes, como o modelo espiral. Infelizmente, o MILSPECs que o 2167A faz referência e os exemplos ilustrativos que ele usa ainda são orientados a cascata, portanto, a maioria das aquisições continuou a usar a cascata, relata Boehm. Uma Força-Tarefa do Conselho de Ciência da Defesa, comandada por Larry Druffel e George Heilmeyer, em seu "Relatório da força-tarefa DSB sobre a aquisição comercial de software de defesa" de 1994, defendeu o uso de modelos mais modernos no atacado.
10. Mills, HD, "Top-down programming in large systems," in *Técnicas de depuração em grandes sistemas*, R. Rustin, ed. Englewood Cliffs, NJ: Prentice-Hall, 1971.
11. Parnas, DL, "On the design and development of program family," *IEEE Trans, em Engenharia de Software*, SE-2, 1 (março de 1976), pp. 1-9; Parnas, DL, "Projetando software para facilitar a extensão e contração," *IEEE Trans, em Engenharia de Software*, SE-5, 2 (março de 1979), pp. 128-138.
12. D. Harel, "Mordendo a bala de prata", *Computador* (Jan., 1992), pp. 8-20.
13. Os artigos seminais sobre ocultação de informações são: Parnas, DL, "Aspectos de distribuição de informações da metodologia de design", Carnegie-Mellon, Dept. of Computer Science, Tech-

- Relatório físico (fevereiro de 1971); Parnas, DL, "Uma técnica para especificação de módulo de software com exemplos," *Com. ACM*, 5, 5 (maio de 1972), pp. 330-336; Parnas, DL (1972). "Sobre os critérios a serem usados na decomposição de sistemas em módulos," *Com. ACM*, 5,12 (dezembro de 1972), pp. 1053-1058.
14. As idéias de objetos foram inicialmente esboçadas por Hoare e Dijkstra, mas o primeiro e mais influente desenvolvimento delas foi a linguagem Simula-67 de Dahl e Nygaard.
 15. Boehm, BW, *Economia da Engenharia de Software*, Englewood Cliffs, NJ: Prentice-Hall, 1981, pp. 83-94; 470-472.
 16. Abdel-Hamid, T. e S. Madnick, *Software de dinâmica de projeto: uma abordagem integrada*, CH. 19, "Aprimoramento de modelo e lei de Brooks." Englewood Cliffs, NJ: Prentice Hall, 1991.
 17. Stutzke, RD, "A Mathematical Expression of Brooks's Law." No *Nono Fórum Internacional sobre COCOMO e Modelagem de Custos*. Los Angeles: 1994.
 18. DeMarco, T. e T. Lister, *Peopleware: Projetos e equipes produtivas*. Nova York: Dorset House, 1987.
 19. Pio XI, Encíclica *Quadragesima Anno*, [Ihm, Claudia Carlen, ed., *As Encíclicas Papais 1903-1939*, Raleigh, NC: McGrath, p. 428.]
 20. Schumacher, E. E, *Pequeno é bonito: economia como se as pessoas importassem*, Edição de biblioteca perene. Nova York: Harper and Row, 1973, p. 244.
 21. Schumacher, op. cit., p. 3. 4.
 22. Um pôster de parede instigante proclama: "A liberdade de imprensa pertence a quem a tem."
 23. Bush, V., "Que possamos pensar," ^ *Atlantic Monthly*, 176, 1 (abril de 1945), pp. 101-108.
 24. Ken Thompson, da Bell Labs, inventor do Unix, percebeu desde cedo a importância das telas grandes para a programação. Ele inventou uma maneira de obter 120 linhas de código, em duas colunas, em seu tubo de armazenamento de elétrons Tektronix primitivo. Agarrei-me a este terminal por meio de toda uma geração de tubos rápidos e de pequenas janelas.

Índice

- Abdel-Hamid, T., 308
tipo de dados abstratos, 188, 220,
273
acidente, 179.182, 209, 214, 272,
280, 281, 303, *viii*
contabilidade, 132
Ada, 188, 283
administrador, 33
Adobe Photoshop, 281
avanço, escada dupla **do**,
119, 242
conselheiro, teste, 192
Aiken, HH, 291 metáfora de
assento de avião, 194 Algol, 34,
44, 64, 68, 203, 295,
302
algoritmo, 102, 239
 alocação, memória dinâmica, 57
 teste alfa, 142, 245, 266
 versão alfa, 240
 Estação de trabalho pessoal alta,
 260 ANSI, 168, 249
 APL, 64, 98, 136, 175, 203, 302
 Apple Computer, Inc., 264, 306
 Apple Desk Top Bus, 307 Apple
 Lisa, 260
 Apple Macintosh, 255, 258, 264,
 284, 291, 306
AppleScript, 287
arquiteto, 37, 41, 54, 62, 66, 79,
100, 233, 236, 238, 255, 257
arquitetura, 44, 143, 233, 234,
245, 266
arquivo, cronológico, 33
aristocracia, 39, 44, 46
Aristóteles, 209, 303
Aron, J., 90, 93, 237, 297
ARPAnetwork, 78
inteligência artificial, 190, 302
assembler, 132
autoridade, 8, 80, 231, 236
AutoCad, 285
AutoLisp, 285
programação automática, 302
Bach, JS, 47
Backus, JW, 64, 295
Backus-Naur Form, 64
Baker, FT, 36, 294, 300
Balzer, R., 302
Bardain, EF, barreira de
2%, sociológico, 119
Begeman, M., 306
Belady, L., 122, 123, 150, 243,
246, 298, 301
Bell Northern Research, 270

Nota: os numerais em negrito indicam uma discussão relativamente substancial de um tópico.

- Bell Telephone Laboratories, 90, 119, 133, 137, 142, 158, 237, 293, *XI*, 308
 Bell, CG, 64, 296, *viii*
 Bengough, W., 107
 Bennington, HD, 305
 versão beta, 240
 Bíblia, 255
 Bierly, R., *viii*
 Blaauw, GA, 45, 49, 62, 63, 294
 Bloch, E., *eu*
 Blum, B., 210
 Boehm, BW, 217, 237, 273, 283, 303, 306, 308, *viii*
 Boehm, EM, 295
 Boes, H., *ix*
 Bohl, M., 302
 Bohm, C., 144, 300
 Booch, G., 302
 Boudot-Lamotte, E., 40
 bala de latão, 219
 descoberta, 186
 Breughel, P., the Elder, 73
 Brooks's Law, 25, 274
 Brooks, FP Jr., 102, 226, 229, 237, 294, 297, 300, 301, 303, i
 Brooks, KP, 216, 224, 306, *viii*
 Brooks, NG, *v*, *viii*
 Buchanan, B., *viii*
 Buchholz, W., 294
 orçamento, 6.108, 110, 239
 acesso, 99, 238
 tamanho, 100, 238
 bug, 142, 143, 195, 209, 231, 235, 242, 243, 244, 245, 272
 documentado, 148
 Abordagem de construção todas as noites, 270
 construção, incremental, 270
 sistema, 147, 246, 301
 estratégia de construção de acordo com o orçamento, 268
 construção, mão de obra, 179
 construindo um programa, 200
 bala, latão, 219
 prata, 179, 207, 212, 214, 226, 303, 304
 Burke, E., 253
 Burks, AW, 194
 Burris, R., *xii*
 Bush, V., 281, 291, 308
 Butler, S., 229
 comprar versus construir, 197
 C++, 220, 285.305
 Cambridge Multiple-Access Sistema, 298
 Cambridge University, 133
 Campbell, E., 121, 242, 298
 Canova, A., 153
 Capp, A., 80
 Carnegie-Mellon University, 78
 declaração CASE, 144
 Case, RP, *viii*, *xi*
 Cashman, TJ, 169
 catedral, 41
 resumo da mudança, 77, 78
 mudança, 117
 controle de, 149
 design, 166, 241, 298
 organização, 118
 mutabilidade, 117, 184, 241
 canal, 45
 engenharia química, 116, 287, 288
 programador chefe, 32, 232
 ClarisWorks, 219
 classe, 189, 222, 225, 272
 Clements, PC, 307, 308
 escriturário, programa, 33
 sistema cliente-servidor, 282
 Clingen, CT, 298, 299
 COBOL, 199, 203, 218
 Codd, E. R., 146, 300

- Jogos CodingWar**, 276
codificação, 20, 237
Coggins, JM, 220, 221, 305, *viii*
Coleman, D., 306
tecla de comando, 263
comando, 261, 286, 287,
comentário 306, 172, 249
comitê, 17, 74, 79
comunicação, 16, 17, 35, 54,
61, 73, 78, 79, 88, 100, 111,
183.232.233.234.235.236,
240, 274
compatibilidade, 63, 64, 68
operação em tempo de compilação,
66 compilador, 132
complexidade, 182, 211, 226, 233,
288
arbitrário, 184, 211
210 conceitual
depuração de componente, 144
componente, 223, 230, 239, 284,
286
manequim 148
compreensibilidade, 186
instalação de computador, 128
construção conceitual, 182, 186,
209
integridade conceitual, 35, 36, 42,
62, 80, 142, 184, 232, 233,
255, 257, 260, 264
estrutura conceitual, 180
conferências, 66
conformidade, 184
Conger, SA, 214, 304
Conklin, J., 259, 306
programa de controle, 91, 93
convergência de depuração,
9
Conway, ME, III, 297
Conway, RW, 47, 294
Cooley, JW, 102
copiloto, 32
Coqui, H., 217, 305
Corbatd, FJ, 93, 146, 237, 293,
297, 298, 299, 300, *XI*
Cornell University, 47
Cosgrove, J., 117, 118, 241, 298
custo, 6, 16, 87, 121, 182, 224, 233,
242, 274
custo, desenvolvimento, 198
carregado frontalmente, 221
coragem, gerencial, 12, 21, 119,
153, 221, 242, 274
tribunal, para disputas de
design, 66
Cox, BJ, 210, 212, 304
Crabbe, G., 163
criação, etapas de componentes, 15,
45, 143
alegria criativa, 7, 120, 280
estilo criativo, 47
trabalho criativo, 46
criatividade, 278, 280
cronograma do caminho crítico, 89.156,
158, 247, 301
Crockwell, D., 87
Crowley, WR, 132
cursor, 261
personalização, 219
personalização, 222
d'Orbais, J., 41
Dahl, OJ, 300, 308
Daley, R. C., base de
dados 300, 108
serviço de dados, 131
banco de dados, 198, 223, 240, 283, 285
tipo de dados, resumo, 189
data, estimada, 158
programado, 158
auxiliar de depuração, 128
depuração, componente, 144
linguagem de alto nível, 135
interativo, 34, 136, 146, 245

- depuração (*contínuo*)
na máquina, 145 •
natureza sequencial de, 17
sistema, 147
- DEC PDP-8, 64
- DEC VMS sistema operacional, 284
- DECLARE, 174
- Força-Tarefa do Defense Science Board
em software militar, *i*, *vii*,
viii
- Defense Science Board, 307
- DeMarco, T., 218, 223, 224, 276,
283, 308, *viii*
- democracia, 44
- Departamento de Defesa, 266
- confiabilidade de depuração
veículo, 131
- Descrição; *Ver* especificação,
mudança de projeto, 241
- design para mudança, 272
- designer, ótimo, 180, 202, 304 metáfora
de área de trabalho, 194, 260, 262
- desenvolvimento, incremental,
diagrama 200, 216
- diferença de julgamento, 35
- diferença de interesse, 35
- Digitek Corporation, 102
- Dijkstra, EW, 144, 300, 308
- diretor, técnico, função de, 79,
236.256
- disciplina, 46, 54, 55, 233
- DiskOperating System, IBM
1410-7010, 56, 57, 99
- terminal de exibição, 78, 129
- divisão de trabalho, 79, 236
- DO ... ENQUANTO, 144
- documento, 107, 239
- sistema de documentação, 134,
244 documentação, 6, 32, 33, 122,
164.224.235.248
- DOD-STD-2167, 266, 307
- DOD-STD-2167A, 307
- Sistema de reconhecimento de voz Dragon,
264
- Druffel, L., 307
- escada dupla de avanço, 119,
242
- componente fictício, 148
- despejo, memória 133, 145
- Durfee, B., 291
- facilidade de uso, 43, 98, 255, 258, 260,
262.263
- Eastman Kodak Company, 285
- Eckman, D., 298
- editor, descrição do trabalho para, 33
text, 32, 34, 68, 128, 133, 134, 146
- Einstein, A., 213
- correo eletrônico, 234, 235
- caderno eletrônico, 78, 235 Sistema
de comutação eletrônico,
encapsulamento 90, 78, 220, 236,
271 Engelbart, D. C, 78, 260, 306
- Inglês, W., 306
- entropia, 122, 243
- meio ambiente, 6, 165.196
- Erikson, WJ, 29, 30, 88, 294,
296
- Ershov, AP, 293, *XI*
- Eschapasse, M., 294
- essência, 179,181,196, 214, 222,
272, 285, 303, *viii*
- estimar, 14, 21, 88, 109, 155,
231, 237, 239, 247, 301
- Evans, BO, *v*
- Everett, RR, 305
- Excel, 285, 286
- sistema especialista, 191
- extensão, 221, 268, 302

- Fagg, P., 24
 Falkoff, AD, 296
 família, produto de software, 268
 Farr, L., 88, 296
 Fast Fourier Transform, 102
 featuritis, 257
 Feigenbaum, EA, 191
 Ferrell, J., 287
 arquivo, fictício, 148
 miniatura, 148
 filtros, 187
 Fjelstadt, 304
 espaço no chão, 239
 flecha de fluxo, 174
 fluxograma, 167, 185, 194, 248, 300
 previsão, 109, 239
 definição formal, 63, 234
 documento formal, 111
 progressão formal de liberação, 133
 formalidade, de propostas escritas, 67
 Fortran, 45, 102, 203, 302
 Fortran, H., 99
 Banco de dados FoxPro, 287
 Franklin, B. (Pobre Richard), 87
 Franklin, JW, 134
 dados de frequência, 306
 suposição de frequência, 257, 259
 fusão, 277

GallopingGertie, Tacoma NarrowsBridge, 264
 Gráfico de Gantt, 265
 General ElectricCompany, gerador 216, 193, 283
 gIBIS, 259, 306
 Ginzberg, MG, 301
 Glass, RL, 214, 226, 304, 305
 Glegg, GL, 294
 Sistema de Posicionamento Global, 257

IR PARA, 170
 Deus, 42, 184, 232, 289, 291, £ x
 Gödel, 213
 Goethe, JW von, 163
 Gold, MM, 146, 246, 300
 Goldstine, HH, 170, 194, 301
 Gordon, P., *mim*

 GOTO, 300
 Grafton, RB, 302
 Grant, EE, 29, 30, 88, 294, 296 \graph, 185, 216
 \structure, 248
 programação gráfica, 194, 302
 ótimo designer, 180, 202, 284
 Greenwald, ID, 295
 software crescente, 180, 200, 212, 268
 Gruenberger, R, 147, 294, 300

 Hamilton, F., 291
 Hamlen, 304
 hardware, computador, 181
 Hardy, H., 97
 Harel, DL, 212, 214, 270, 304, 305, 307
 Harr, J., 90, 93, 137, 237, 299, XI
 Hayes-Roth, R., *viii*
 Heilmeyer, G., 307
 Heinlein, RA, 81, 296
 Hennessy, J., 306
 Henricksen, JO, 299
 Henry, P., 253
 Herzberg, F., 210, 304
 Hetzel, W. C, 299
 Hezel, KD, 301
 estrutura hierárquica, 189, 212, 220
 linguagem de alto nível, *Ver*
 linguagem de alto nível
 Hoare, CAR, 300, 308

- Homer, 255
 Hopkins, M., 299
 Huang, W., 222, 305
 hustle, 155, 247
 HyperCard, 285
 hipertexto, 281, 291, 306
- IBM 1401, 45, 65, 130
 IBM650, 43, 102
 IBM 701, 131
 IBM7030Stretchcomputer, 44,
 47, 55, 291, 300, *eu*
 IBM 704, 55
 IBM 709, 55, 57, 181
 IBM7090, 55, 64
 IBMCorporation, 90, 119, 291, *vii*
 Computador IBMHarvest, *eu*
 Sistema operacional IBMMVS / 370,
 276, 284
 IBMOperatingSystem / 360, *Ver*
 Sistema Operacional / 360
 Sistema operacional IBMOS-2,
 284 IBMPCCcomputer, 260, 264,
 284 IBMSAGEANFSQ / 7data
 sistema de processamento, 216, 305
 família de computadores IBMSystem / 360
 44, 45, 62, 64
 IBMSystem / 360Model 165, 98
 IBMSystem / 360Model 30, 45, 47
 IBMSystem / 360Model 65, 99
 IBMSystem / 360Model75, 47
IBM. Princípios do Sistema / 360 de
 Operação, 62
 IBMVM / 360operatingsystem,
 284
 IBSYSoperatingsystemfor the
 7.090, 56
 Ichikawa, T., 302
 ícone 260
 ideias, como estágio de criação,
 15 IEEE *Computador* revista, *vii*
- SE ... ENTÃO ... OUTRO, 144
 Ihm, C. C, 308
 implementação, 15, 45, 64, 143,
 209, 233, 234, 238, 256, 266
 implementações, múltiplas, 68
 implemento, 47, 54, 62, 66
 incorporação, direta, 66, 118,
 241, 264
 desenvolvimento incremental, 200,
 268
 modelo de construção incremental, 212,
 267.270
 recuo, 174
 ocultação de informação, 78, 271, 308
 teoria da informação, 212
 herança, 220, 222, 273
 inicialização, 174
 faixa de entrada, 6.165, 248
 formato de entrada-saída,
 165 instrumentação, 129
 integridade, conceitual, 35, 36, 42,
 43, 62, 80, 142, 255, 257, 260,
 264
 interação, como parte da criação,
 15, 209
 primeiro da sessão, 146
 depuração interativa, 34.146
 programação interativa, 136,
 244, 245, 246
 interface, 6, 32, 62, 66, 79, 118,
 120, 122, 241, 243, 255, 257,
 264, 271, 282, 286, 306
 metaprogramação, 287
 módulo, 268
 WIMP, 234, 260, 263
 Interlisp, 187
 International Computers Limited,
 89, 133, 293, *XI*
 Internet, 306
 intérprete, para economia de espaço,
 102 invisibilidade, 185, 216, 241

- iteração, 199, 267
 Iverson, KE, 64, 102, 170, 291, 294, 295, 296, 297, 300, 301, 302
 Jacopini, A., 144, 300
 Jobs, S., 260
 Jones, C., 217, 218, 222, 223, 224, 305
 alegrias do ofício, 7
 Kane, M., 295
 teclado, 262
 Chaves, WJ, 169 King, WR, 301 Knight, CR, 3 Knuth, DE, 102, 297
 Kugler, HJ, 303
 etiqueta, 174
 Lachover, H., 305
 Lake, C., 291
 Landy, B., 298
 Lang, DE, 302
 descrição da linguagem, formal, 181
 tradutor de linguagem, 93
 linguagem, quarta geração, 283
 de alto nível, 118.135, 143, 146, 186, 194, 225, 237, 241, 244, 245, 248, 249
 máquina, 180, 225
 programação, 68, 180, 186, 225 script, 287
 projeto atrasado, 13, 89, 217, 235, 246, 275, 306
 advogado, linguagem, 34
 Lehman, M., 122, 123, 150, 243, 246, 298, 301
 Lewis, CS, 123, 298 library, 187, 222, 239, 244, 272, 305
 classe, 225
 macro, 34
 programa, 132
 editor de ligação, 56, 282 Lister, T., 276, 308
 Little, AD, 287
 Locken, OS, 76
 Lowry, ES, 300
 Lucas, P., 295
 Lukasik, S., 211
 Interface MacintoshWIMP, 234, 260, 263
 Madnick, S., 274, 308
 mágica, 7, 142, 226
 manutenção, 118, 242
 homem-mês, 16, 231, 273 sistema de informação de gestão (MIS), 107, 111, 219, 239, 240, 285
 manual, 62, 239, 258, 263
 Sistema / 360, 62
 mercado, massa, 180, 197, 218, 223, 258
 gestão matricial, 222
 organização do tipo matriz, 79
 Mausner, B., 210, 304
 Mayer, DB, 297
 McCarthy, J., 247, 270, 278, 306, através da
 McCracken, DD, 302
 McDonough, E., 300
 Mealy, G., XI
 medição, 222
 meio de criação, tratável, 7, 15, 117
 reunião, ação do problema, 157
 revisão de status, 75, 157
 padrão de uso de memória, 129, 239
 mentor, 203, 275
 menu, 260

- Merwin, RE, 299
Merwin-Dagget, M., 300
metáfora, 260
metaprogramação, 285
revolução do microcomputador, 214,
 279
microficha, 77, 235
Microsoft Corporation, 246, 270
Microsoft Windows, 260
Microsoft Word 6.0, 258, 306
MicrosoftWorks, 219
marco, 22, 25, 154, 158, 247,
 248, 270
Mills, HD, 32, 33, 201, 267,
 271, 294, 299, 300, 303, 307
MILSPECdocumentação, 248
mini-decisão, 63, 111, 234, 240
MiniCad design program, 285
MIT, 93, 121, 146, 287, 293, *XI*
nome mnemônico, 174
modelo, 255.256.274.307
 COCOMO, 273
 construção incremental, 212, 267,
 270
 espiral, 303.307
 cachoeira, 264, 307
Modula, 189, 203
modularidade, módulo 118, 188,
220, 101, 122, 143, 241, 243,
 245, 269, 271, 273, 285
módulos, número de,
122
Mooers, CN, 44
Moore, SE, *xii*
Morin, LH, 88, 296
Mostow, J., 302
mouse 307
projetos de mudança, 277
Mozart, WA 202
MS-DOS, 203, 255, 284 Multics,
93, 136, 146, 237, 298,
- 299.300
múltiplas implementações, 68
MVS / 370, 203
Naamad, A., 305
Nanus, B., 88, 296
Naur, P., 64
Needham, RM, 298
Nelson, EA, 297
aninhamento, como auxílio de documentação,
 172
natureza da rede de
 comunicação, 79
Neustadt, RE, 295
Newell, A., 64, 296
Noah, 97
Universidade Estadual da Carolina do Norte,
 287
status do notebook 33
 sistema, 147
Nygaard, 308
objeto 285
design orientado a objetos, 302
programação orientada a objetos,
 189, 219, 273
objetivo, 8, 75.108.110, 117, 239
 custo e desempenho, 49
 espaço e tempo, 49
obsolescência, 9.26.123
pacote de prateleira, 198
espaço de escritório, 242
Ogden, JL, 293, 298
sistema aberto, 283
sistema operacional, 128, 238, 243,
 283
Sistema operacional / 360, 43, 45, 47,
 56, 76, 93, 129, 234, 235, 237,
 243, 271, 276, 295, *i, x*
otimismo, 14, 212, 231
opção, 101, 165, 238
Orbais, J. d ', 41

- ordem de grandeza
melhoria, 208, 213, 215,
281, 291, *vii*
- organograma, 108, 111, 239
- organização, 74, 78.118, 235,
236.242
- OS / 360 *Conceitos e instalações*, 134
- OS / 360Queued
Método de acesso de
telecomunicações, 285
- OS / 360, *Ver Sistema operacional /*
360
- sobreposição, 54, 99, 129
- visão geral, 165, 248
- Ovídio, 55
- Padegs, A., 62
- papelada 108
- Famílias Parnas, 268
- Parnas, DL, 78, 190, 193, 212,
221, 224, 226, 236, 268, 271,
288, 296, 302, 304, 307, 308,
através da
- particionamento, 16, 231
- Linguagem de programação Pascal,
203, 285
- Pascal, B., 123
- estrutura de passagem, 166
- Patrick, RL, *vii*
- Patterson, D., 306
- pessoas, 29, 202.276, 284
- Peopleware: Projetos Produtivos e*
Equipes, 276
- perfeição, requisito para, 8
- simulador de desempenho, 134
- desempenho, 182, 258
- Gráfico PERT, 89, 156, 158, 247
- pessimismo, 212
- Pedro, o Apóstolo, 171
- pedra filosofal, 304
- Piestrasanta, AM, 160, *XI*
- planta piloto, 116.240
- sistema piloto, 298
- tubos, 187
- Pisano, A., 127
- PiusXI, 277, 308
- PL / Clanguage, 47, 294
- PL / I, 32, 47, 64, 66, 93, 135, 172,
203, 245, 299, 302
- planejamento, 20
- Organização de planos e controles,
160, 248
- cercadinho, 133, 149, 244, 246
- Pnueli, A., 305
- apontando, 260
- sistema policiado, 65
- Politi, M., 305
- Pomeroy, J. W., 298
- Pobre Richard (Benjamin
Franklin), 87
- Papa Alexandre, 207
- Portman, C, 89, 237, 293, 296, *xi*
- ferramentas de poder para a
mente, 219 poder, desistir, 277
- prática, bom software
engenharia, 193, 202
- preço, 109
- PROCEDIMENTO, 174
- procedimento, catalogado, 34
- produtor, função de, 79, 236, 256
- teste de produto, 69, 142, 234, 245
- produto, emocionante, 203
- sistema de programação, 4, 230
- programação, 5, 288
- equação de produtividade, 197
- produtividade, programação, 21,
30, 88, 94, 135, 181, 186, 208,
213, 217, 237, 244, 245, 273,
276, 284
- programclerk, 33
- biblioteca de programas, 132
- manutenção de programação, 120

- nome do programa, 174
produtos do programa, 273
gráfico da estrutura do programa, 170,
 185
programa, 4
 auxiliar, 149
 autodocumentado, 171
retreinamento do programador, 220,
221 ambiente de programação, 187
linguagem de programação, 221
produto de programação, 5, 116,
 240
sistema de programação, 6
produto de sistemas de programação,
 4, 230
projeto de sistemas de programação,
 237
programação, automática,
 193 gráfica, 194
 visual, 194
refinamento progressivo, 267, 299
ProjectMercuryReal-Time
 Sistema, 56
pasta de trabalho do projeto, 235
promoção, na classificação, 120
prototipagem, rápida, 180, 199, 270
Publilius, 87
técnica do fio roxo, 149 objetivo, de
um programa, 165, 249 de uma
variável, 174
- Quadragesima Anno, Encíclica,*
 277.308
qualidade, 217
quantização, de mudança, 62, 118,
 150.246
 de demanda por mudança, 117
- Raeder, G., 302
aumento de salário, 120
- Ralston, A., 300
prototipagem rápida, 180, 199, 270
sistema em tempo real, 301
realismo, 212, 226
realização, etapa da criação, 49,
 143, 256, 266
refinamento, progressivo, 143,
 267, 299
 requisitos, 199
desastre programado regenerativo, 21
Catedral de Reims, 41
lançamento, programa, 121, 185, 217,
 243, 244
confiabilidade, 186
entrada de trabalho remoto, 58
reparticionamento, 24, 232, 275
representação, de informações,
 102, 239
refinamento de requisitos, 199
reprogramação, 24
responsabilidade, versus autoridade,
 8.231
Restaurante Antoine, 13 /
componente reutilizável, 210 /
reutilização, 222, 224, 269, 273,
285 Reynolds, CH, 294, 301
conflito de funções, redução, 157
ROM, memória somente leitura,
234 Roosevelt, FD, 115, 298
Rosen, S., 300
Royce, WW, 265, 307
Rustin, R., 300, 303,
307 Ruth, GH (Babe), 87
- Sackman, H., 20, 29, 88, 294, 296
Salieri, A., 202
Saltzer, JH, 298, 299
Sayderman, BB, 210, 304
Sayers, DL, 15, 209, 303
andaimes, 34, 148, 246

- scalingup, 36,116, 240, 288
 Scalzi, C A., 300
 cronograma, 79,108, 111, 154, 239,
 244, 247, 265, 273, *Ver Projeto*
 atrasado
 custo ótimo, 274
 agendador, 57
 programação, 14, 129
 Schumacher, EF, 277, 279,
 308
 tela, 194, 201, 268, 287, 308 efeito
 do segundo sistema, 51, 234,
 257.259
 secretária, 33
 segurança, 183
 programa de autodocumentação, 118,
 171, 249
 Selin, I., 219
 semântica, 44, 64, 66, 225, 261,
 272
 Shakespeare, W., 141, 255
 Shannon, EC, 212
 Compartilhe 709 sistema operacional
 (SOS), 295
 Compartilhar sistema operacional para o
 IBM709, 57
 Shell, DL, 295
 Sherman, M., 189
 Sherman, R., 305
 atalhos, 263
 software encolhido, 218,
 219, 223, 257, 279, 282, 284
 Shtul-Trauring, A., 305 efeito
 colateral, 65, 122, 243
 bala de prata, 179, 207, 212, 214,
 226, 303, 304
 simplicidade, 44, 181, 186
 Simula-67, 189, 308
 simulador, 234, 244
 meio ambiente, 131
 lógica, 65, 131
 desempenho, 99
 tamanho, programa, 30, 98,129,135,
 175
 Skwiersky, BM, 303 slippage,
 cronograma, ver tarde
 projeto
 Sloane, JC, *xii*
Pequeno é Bonito, 277
 Smalltalk, 203, 220
 Smith, S., 97
 instantâneo, 145
 Snyder, Van, 222
 barreira sociológica, 119
 Sedahl, EU, 211
Economia da Engenharia de Software,
 273
 Instituto de Engenharia de Software,
 202
 indústria de software, 282
Software de dinâmica de projeto,
 274 Sófocles, 153.155
 alocação de espaço, 108, 111
 espaço, memória, 238
 escritório, 203, 276
 programa, *Vertamanho*, programa,
 especialização de função, 35, 79,
 236
 especificação, 195, 200, 245, 256,
 266, 301, 304
 arquitetônico, 43, 142, 245
 funcional, 32, 48, 49, 62, 75,
 108, 110
 interface, 75
 interno, 75
 desempenho, 32
 testando o, 142
 velocidade, programa, 30, 98.135
 espiral, previsão de preços,
 planilha 109, 198, 280, 281

- grupo de funcionários, 79
pessoal, projeto, 21, 273
Stalnaker, AW, 297 padrão, 75, 168, 249, 283 padrão, de facto, 264 Stanford Research Institute, 78, 260
Stanton, N., ix
 empresa inicial, 278, 284
 ferramenta de design Statemate, 305 controle de status, 108
 relatório de status, 157, 247
 reunião de revisão de status, 157
 símbolo de status, 81
Steel, TB, Jr., 295 Strachey, C, 56, 146, 295, 300
 franqueza, 44
 Strategic Defense Initiative, 257
 Stretch Operating System, 56, 99
 programação estruturada, 32.144, 218.241.245
 stub, 267
Stutzke, RD, 275, 308
 sub-rotina, 182, 272
 Função subsidiária, princípio de, 277
 superior-subordinado
 relacionamento 35
 programa de supervisão, 146
 custo de suporte, 218
 equipe cirúrgica, 27, 120, 232 Sussenguth, EH, 296
Swift, J., 115
 sincronismo no arquivo, sintaxe 171, 44, 64, 66, 225, 262 abstrato, 64
 construção do sistema, 147, 246, 301
 depuração do sistema, 132, 147
 Desenvolvimento do sistema Corporation, 88, 297
 sub-biblioteca de integração do sistema, 133
 teste do sistema, 19, 122, 133, 147 sistema, grande, 31
 programação, 6, 288
 Família de computadores System / 360, 296, 301
 produto de sistemas, programação, 4.288
Ponte Tacoma Narrows, 115, 264
Taliaffero, WM, 297
 máquina alvo, 129, 243, 244
Taub, AH, 301
Taylor, B., 260
 equipe, pequena, afiada, 30
 diretor técnico, função de, 79, 236, 256
 tecnologia, programação, 49, 102.128
 registro de telefone, 68, 234
 caso de teste, 6, 34, 147, 166, 192, 248, 270
 teste, componente, 20
 sistema, 19, 122, 133, 147
 testador, 34
 consultor de teste, 192
teste, 6
 regressão, 122, 242,
 especificação 268, 142
 Instalação de depuração TESTRAN, 57, 146
 sistema de edição de texto, 134, 244 Thompson, K., 308
 jogar fora, 116, 241, 264
 tempo, calendário, 14, 231
 máquina, 180
 Sistema de compartilhamento de tempo, PDP-10, 43
 Sistema de Compartilhamento de Tempo / 360, 136, 146

- tempo compartilhado, 187, 280, 282, 300 ferramenta, 125, 196, 243, 258, 280 poder, para a mente, 219
- ferreiro, 34, 128**
- design de cima para baixo, 220, 134, 143, 245
- programação de cima para baixo, 303 Torre de Babel, 72, 235
- TRAClanguage, 44**
- rastreamento, programa 146
- compensação, função de tamanho, velocidade de tamanho 101, 99, 101
- treinamento, tempo para, 18
- área transitória, 101, 238
- Trapnell, FM, 301, *x, xi*
- organização da árvore, 79
- Truman, HS, 61
- TRW, Inc., 273
- Tukey, JW, 102
- tempo de resposta, 136, 187, 245, 281
- volume de negócios, pessoal, 184 Turski, WM, 213, 304
- problema de dois cursores, 261
- operação com duas mãos, 262
- tipo, dados abstratos, 188
- verificação de tipo, 220
- Computador Univac, 215
- Estação de trabalho Unix, 282
- Unix, 187, 197, 203, 244, 284, 287, 308
- Universidade da Carolina do Norte em Chapel Hill, *eu*
- usuário, 45, 117, 121, 165, 255, 258, 261, 266, 268, 271, 286
- novato, 263
- poder, 263
- USSRAcademy of Sciences, XI**
- programa utilitário, 34, 128, 134
- Projeto Vanilla Framework**
- metodologia, 216
 - máquina do veículo, 131
 - verificação, 195
 - versão, 118, 185, 217, 241, 268, 270
 - alfa, 240
 - beta 240
- Vessey, 214
- ambiente virtual, 281, *i, viii***
- memória virtual, 238
- programação visual, 302
- representação visual, 216
- vocabulários, grande, 224
- Reconhecimento de voz do Voice Navigator sistema, 264
- von Neumann, J., 170, 194, 301
- Vyssotsky, VA, 142, 158, 179, 185, 245, 248, 293, 299, *XI*
- Walk, K. 295
- Walter, AB, 302
- Ward, F., *viii*
- modelo em cascata, 264, 307 Watson, TJ, Jr., *v, xi*
 - Watson, TJ, Sr., 164
 - Weinberg, GM, 302
 - Weinwurm, GF, 88, 295, 296, 301
 - Weiss, EA, 303
 - Wells Apocalypse, The*, 61
 - Iobisomem, 178, 180, 208
 - Wheeler, E., 279, *viii*
 - Guilherme III da Inglaterra, Príncipe da Laranja, 207
 - Wilson, TA, 301
 - Interface WIMP, 234, 260, janela 263, 260
 - Sistema operacional Windows NT, 2. 3. 4

- Sistema operacional Windows,
284 Wirth, N., 143, 245, 299, 306
Witterrongel, DM, 301
Técnica mágica de Oz, 271
Wolverton, RW, 293, 297 pasta
de trabalho, 75
posto de trabalho, 196
World-WideWeb, 235, 281
buraco de minhoca, 287
Wright, WV, 167
- Centro de Pesquisa PaloAlto da Xerox,
260
Yourdon, E., 218, 223, 224, 305,
VIII
zoom, 165, 248
Zraket, CA, 305