Sem bala de prata

—Essência e Acidente em Engenharia de Software

Frederico P. Brooks, Jr.
Universidade da Carolina do Norte em Chapel Hill

> *Não existe um desenvolvimento único, seja na tecnologia ou na técnica de gestão, que por si só prometa uma melhoria de uma ordem de grandeza dentro de uma década na produtividade, na fiabilidade, na simplicidade.*

Resumo1

Toda construção de software envolve tarefas essenciais, a formação de 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 destas em linguagens de máquina dentro de restrições de espaço e velocidade. A maior parte dos grandes ganhos anteriores em produtividade de software resultou da remoção de barreiras artificiais que tornaram as tarefas acidentais excessivamente difíceis, tais como restrições severas de hardware, linguagens de programação inadequadas e falta de tempo de máquina. Quanto do que os engenheiros de software fazem hoje ainda é dedicado ao acidental, em oposição ao essencial? A menos que seja mais de 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 cada vez 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 do nosso folclore, nenhum aterroriza mais do que os lobisomens, porque eles se transformam inesperadamente do familiar em horror. Para estes, buscamos balas de prata que possam magicamente colocá-los para descansar.

---

[1] Reproduzido de: Frederick P. Brooks, *The Mythical Man-Month, edição de aniversário com 4 novos capítulos,* Addison-Wesley (1995), ele próprio reimpresso de *Proceedings of the IFIP Tenth World Computing Conference,* H.-J. Kugler, ed., Elsevier Science BV, Amsterdã, NL (1986) pp.

O projeto de software familiar tem algo desse caráter (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 defeituosos. Portanto, ouvimos gritos desesperados por uma solução mágica, 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 solução mágica. Não existe um desenvolvimento único, seja na tecnologia ou na técnica de gestão, que por si só prometa uma melhoria de uma ordem de grandeza na produtividade, na fiabilidade, na simplicidade. Neste capítulo tentaremos ver o porquê, mas examinando tanto a natureza do problema de software quanto as propriedades dos marcadores propostos.

Contudo, o ceticismo não é pessimismo. Embora não vejamos avanços surpreendentes e, na verdade, acreditemos que sejam inconsistentes com a natureza do software, muitas inovações encorajadoras estão em curso. Um esforço disciplinado e consistente para desenvolvê-los, propagá-los e explorá-los deveria, de facto, produzir uma melhoria de 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 das teorias dos humores pela teoria dos germes. Esse mesmo passo, o início da esperança, por si só frustrou todas as esperanças de soluções mágicas. Disse aos trabalhadores que o progresso seria feito gradualmente, com grande esforço, e que um cuidado persistente e incessante teria de ser prestado a uma disciplina de limpeza. O mesmo acontece com a engenharia de software hoje.

Tem que ser difícil? – Dificuldades Essenciais

Não só não existem soluções mágicas à vista, como a própria natureza do software torna improvável que existam quaisquer invenções que façam pela produtividade, confiabilidade e simplicidade do software o que a eletrônica, os transistores e a integração em larga escala fizeram pela hardware de computador. Não podemos esperar ver ganhos duplicados a cada dois anos.

Primeiro, devemos observar que a anomalia não é que o progresso do software seja tão lento, mas que o progresso do hardware do computador seja muito rápido. Nenhuma outra tecnologia, desde o início da civilização, registou um ganho de preço-desempenho de seis ordens de magnitude em 30 anos. Em nenhuma outra tecnologia se pode optar por obter ganhos em termos de melhor desempenho ou redução de custos. Esses ganhos decorrem da transformação da fabricação de computadores de uma indústria de montagem para uma indústria de processamento.

Em segundo lugar, para ver que taxa de progresso podemos esperar na tecnologia de software, examinemos as suas dificuldades. Seguindo Aristóteles, divido-os em essência – as dificuldades inerentes à natureza do software – e acidentes – aquelas dificuldades que hoje acompanham a sua produção, mas que não são inerentes.

Os acidentes que discuto na próxima seção. Primeiro, consideremos 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. Esta essência é abstrata, na medida em que a construção conceitual é a mesma sob muitas representações diferentes. No entanto, é altamente preciso e ricamente detalhado.

Acredito que a parte difícil da construção de software seja a especificação, o design e o teste dessa construção conceitual, e 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 existe inerentemente nenhuma solução mágica.

Consideremos as propriedades inerentes desta essência irredutível dos sistemas de software modernos: complexidade, conformidade, mutabilidade e invisibilidade.

Complexidade. As entidades de software são mais complexas pelo seu tamanho do que talvez qualquer outra construção humana, porque não há duas partes iguais (pelo menos acima do nível de declaração). Se forem, transformamos as duas partes semelhantes em uma só, uma sub-rotina, aberta ou fechada. Neste aspecto, os sistemas de software diferem profundamente dos computadores, edifícios ou automóveis, onde abundam 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 apenas 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 forma não linear e a complexidade do todo aumenta muito mais do que linearmente.

A complexidade do software é uma propriedade essencial e não acidental. Conseqüentemente, as descrições de uma entidade de software que abstraem sua complexidade geralmente abstraem sua essência. A matemática e as ciências físicas fizeram grandes progressos durante três séculos, construindo modelos simplificados de fenómenos complexos, derivando propriedades dos modelos e verificando essas propriedades experimentalmente. Isto 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 derivaram desta complexidade essencial e sua não linearidade aumentou com o tamanho. Da complexidade surge 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, e muito menos compreender, todos os estados possíveis do programa, e daí vem a falta de confiabilidade. Da complexidade das funções surge a dificuldade de invocá-las, o que torna os programas difíceis de usar. Da complexidade da estrutura surge a dificuldade de estender programas a novas funções sem criar efeitos colaterais. Da complexidade da estrutura surge o estado não visualizado que constitui alçapões de segurança.

Não apenas os problemas técnicos, mas também os problemas de gestão decorrem da complexidade. Esta complexidade dificulta a visão geral, impedindo assim a integridade conceitual. Torna difícil encontrar e controlar todas as pontas soltas. Isso cria uma tremenda 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 “fundamental” das partículas. O físico, contudo, continua a trabalhar com a firme fé de que existem princípios unificadores a serem encontrados, seja nos quarks ou nas 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. Grande parte da complexidade que ele deve dominar é uma complexidade arbitrária, forçada sem rima ou razão pelos muitos seres humanos.

instituições e sistemas aos quais suas interfaces devem confirmar. Eles diferem de interface para interface, e de tempos em tempos, não por necessidade, mas apenas porque foram projetados por pessoas diferentes, e não por Deus.

Em muitos casos, o software deve confirmar porque entrou em cena mais recentemente. Noutros, deve conformar-se porque é percebido como o mais conformável. Mas em todos os casos, muita complexidade vem da conformação com outras interfaces; isso não pode ser simplificado apenas por uma reformulação do software.

Mutabilidade. A entidade de software está constantemente sujeita a pressões por mudanças. Claro, o mesmo acontece com edifícios, carros e computadores. Mas as coisas manufaturadas raramente são alteradas após a fabricação; eles são substituídos por modelos posteriores ou alterações essenciais são incorporadas em cópias posteriores com números de série do mesmo design básico. Retornos de chamadas de automóveis são bastante raros; mudanças de campo dos computadores um pouco menos.
Ambos são muito menos frequentes do que modificações no software em campo.

Em parte, isso ocorre porque o software de 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 pura matéria de pensamento, infinitamente maleável. Os edifícios são, de facto, mudados, mas os elevados custos da mudança, compreendidos por todos, servem para atenuar os caprichos dos que mudam.

Todo software bem-sucedido é alterado. 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 funções estendidas vêm principalmente de usuários que gostam da função básica e inventam novos usos para ela.

Em segundo lugar, o software bem-sucedido também sobrevive além da vida normal do veículo-máquina para o qual foi escrito inicialmente. Se não forem novos computadores, pelo menos surgirão novos discos, novos monitores, novas impressoras; e o software deve estar em conformidade com seus novos veículos de oportunidade.

Em suma, o produto de software está incorporado numa matriz cultural de aplicações, utilizadores, leis e veículos mecânicos. Todos eles mudam continuamente e suas mudanças forçam inexoravelmente mudanças no produto de software.

Invisibilidade. O software é invisível e não visualizável. Abstrações geométricas são ferramentas poderosas. A planta baixa de um edifício ajuda tanto o arquiteto quanto 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, embora sejam abstrações, servem ao mesmo propósito. Uma realidade geométrica é capturada em uma abstração geométrica.

A realidade do software não está inerentemente incorporada no espaço. Portanto, não há 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 diversos gráficos podem representar o fluxo de controle, o fluxo de dados, padrões de dependência, sequência temporal, relações nome-espaço. Geralmente nem são planares, muito menos hierárquicas. Na verdade, uma das formas de

estabelecer controle conceitual sobre tal estrutura é impor o corte de links até que um ou mais gráficos se tornem hierárquicos.2

Apesar do progresso na restrição e simplificação das estruturas de software, elas permanecem inerentemente não visualizáveis, privando assim a mente de algumas das suas ferramentas conceituais mais poderosas. Esta falta não só impede o processo de design dentro de uma mente, como também dificulta gravemente a comunicação entre mentes.

Avanços anteriores resolveram dificuldades acidentais Se examinarmos as três etapas da tecnologia de software que foram mais frutíferas no passado, descobriremos que cada uma atacou uma grande dificuldade diferente na construção de software, mas foram dificuldades acidentais, e não essenciais. Também podemos ver os limites naturais da extrapolação de cada um desses ataques.

Linguagens de alto nível. Certamente o golpe mais poderoso para a produtividade, confiabilidade e simplicidade do software foi o uso progressivo de linguagens de alto nível para programação. A maioria dos observadores atribui a esse desenvolvimento pelo menos um factor de cinco em produtividade, e com ganhos concomitantes em fiabilidade, 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 preocupa-se 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, ela elimina todo um nível de complexidade que nunca foi inerente ao programa.

O máximo que uma linguagem de alto nível pode fazer é fornecer todas as construções que o programador imagina no programa abstrato. É certo que o nível da nossa sofisticação na reflexão sobre estruturas de dados, tipos de dados e operações está a aumentar constantemente, mas a um ritmo cada vez menor. E o desenvolvimento da linguagem se aproxima cada vez mais da sofisticação dos usuários.

Além disso, em algum momento a elaboração de uma linguagem de alto nível torna-se um fardo que aumenta, e não reduz, a tarefa intelectual do usuário que raramente utiliza as construções esotéricas.

Compartilhamento de tempo. A maioria dos observadores atribui ao compartilhamento de tempo uma grande melhoria na produtividade dos programadores e na qualidade de seus produtos, embora não tão grande quanto a trazida pelas 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 reviravolta da programação em lote significa que inevitavelmente esquecemos as minúcias, se não a própria essência, do que estávamos pensando quando paramos a programação e solicitamos a compilação e a execução. Esta interrupção da consciência custa tempo, pois precisamos nos refrescar. O efeito mais grave poderá muito bem ser a deterioração da compreensão de tudo o que se passa num sistema complexo.

---

[2] Parnas, DL, “Projetando software para facilidade de extensão e contração”, *IEEE Trans. em SE,* 5, 2 (março de 1979), pp.

O retorno lento, assim 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 chega a zero, em algum momento ultrapassa o limite humano de notabilidade, cerca de 100 milissegundos.
Além disso, nenhum benefício é esperado.

Ambientes de programação unificados. Unix e Interlisp, os primeiros ambientes de programação integrados a serem amplamente utilizados, são percebidos como tendo melhorado a produtividade por fatores integrais. Por que?

Eles atacam as dificuldades acidentais de usar programas em conjunto, fornecendo bibliotecas integradas, formatos de arquivo unificados e pilhas e filtros. Como resultado, estruturas conceituais que, em princípio, poderiam sempre chamar, alimentar e usar umas às outras, podem de fato fazê-lo facilmente na prática.

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

Devido a esses sucessos, os ambientes são objeto de grande parte dos softwares atuais.
pesquisa de engenharia. Veremos suas promessas e limitações na próxima seção.

Esperanças pela Prata Agora

vamos considerar os desenvolvimentos técnicos que são mais frequentemente avançados como potenciais balas de prata. Que problemas eles abordam? Serão problemas de essência ou serão restos de nossas dificuldades acidentais? Eles oferecem avanços revolucionários ou incrementais?

Ada e outros avanços linguísticos de alto nível. Um dos desenvolvimentos recentes mais elogiados é a linguagem de programação Ada, uma linguagem de uso geral e de alto nível da década de 1980. Na verdade, Ada não apenas reflete melhorias evolutivas nos conceitos de linguagem, mas também incorpora recursos para encorajar conceitos modernos de design e modularização. Talvez a filosofia Ada seja um avanço maior do que a linguagem Ada, pois é a filosofia da modularização, dos tipos de dados abstratos, da estruturação hierárquica.

Ada talvez seja excessivamente rica, o produto natural do processo pelo qual foram impostos requisitos ao seu design. Isso não é fatal, pois os vocabulários de trabalho em subconjuntos podem resolver o problema de aprendizagem, e os avanços de hardware nos darão o MIPS barato para pagar os custos de compilação. Avançar na estruturação de sistemas de software é de fato um uso muito bom para o aumento do MIPS que nosso dinheiro comprará. Os sistemas operacionais, fortemente 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 aumento de hardware anterior.

No entanto, Ada não provará ser a solução mágica que destruirá o monstro da produtividade do software. Afinal, trata-se apenas de mais uma linguagem de alto nível, e a maior recompensa de tais linguagens veio da primeira transição, das complexidades acidentais da máquina para a declaração mais abstrata de soluções passo a passo.
Uma vez eliminados esses acidentes, os restantes serão menores e o retorno da sua remoção será certamente menor.

Prevejo que daqui a uma década, quando a eficácia da Ada for avaliada, verificar-se-á que esta fez uma diferença substancial, mas não por causa de qualquer característica linguística específica, nem mesmo por causa de todas elas combinadas. Nem a nova Ada

ambiente provam ser a causa das melhorias. A maior contribuição de Ada será que a mudança para ela ocasionou o treinamento de programadores em técnicas modernas de design de software.

Programação orientada a objetos. Muitos estudantes da área têm mais esperança na programação orientada a objetos do que em qualquer outra moda técnica da época.3 Estou entre eles. Mark Sherman, de Dartmouth, observa que devemos ter o cuidado de distinguir duas ideias distintas que recebem esse nome: tipos de dados abstratos e tipos hierárquicos, também chamados de classes. O conceito do tipo de dados abstrato é que o tipo de um objeto deve ser definido por um nome, um conjunto de valores próprios 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 do Modula.

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 ocultação e ocultação sem hierarquias.
Ambos os conceitos representam avanços reais na arte de construir software.

Cada um remove mais uma dificuldade acidental do processo, permitindo ao designer expressar a essência do seu design sem ter que expressar grandes quantidades de material sintático que não acrescenta nenhum novo conteúdo de informação. Tanto para tipos abstratos quanto para 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 design em si é essencial; e tais ataques não alteram em nada isso. Um ganho de ordem de grandeza só pode ser obtido pela programação orientada a objetos se a vegetação desnecessária de especificação de tipo que permanece hoje em nossa linguagem de programação for responsável por nove décimos do trabalho envolvido no projeto de um produto de programa. Eu duvido.

Inteligência artificial. Muitas pessoas esperam que os avanços na inteligência artificial proporcionem o avanço revolucionário que proporcionará ganhos de ordem de grandeza na produtividade e qualidade do software.4 Não acredito. Para perceber porquê, temos de dissecar o que se entende por “inteligência artificial” e depois ver como se aplica.

Parnas esclareceu o caos terminológico:

Duas definições bastante diferentes de IA são de uso comum hoje. AI-1: O uso de computadores para resolver problemas que antes só poderiam ser resolvidos com a aplicação da inteligência humana. AI-2: O uso de um conjunto específico de técnicas de programação conhecido como programação heurística ou baseada em regras. Nesta abordagem, especialistas humanos estudam 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. . . O

[3] Booch, G., “Design orientado a objetos”, em *Engenharia de Software com Ada.* Menlo Park, Califórnia: Benjamin Cummings, 1983.

[4] Mostow, J., ed., Edição Especial sobre Inteligência Artificial e Engenharia de Software, *IEEE Trans. em SE,* **11,** 11 (novembro de 1985).

> *A primeira definição tem um significado deslizante. . · . Algo pode se enquadrar hoje na definição de AI-1, mas, uma vez que vejamos como o programa funciona e entendamos o problema, não pensaremos mais nele como IA. . . . Infelizmente não consigo identificar um conjunto de tecnologia que seja exclusivo para este campo. . . . A maior parte do trabalho é específica do problema e é necessária alguma abstração ou criatividade para ver como transferi-lo.5*

Concordo plenamente com esta crítica. As técnicas utilizadas para reconhecimento de fala parecem ter pouco em comum com aquelas utilizadas para reconhecimento de imagens, e ambas são diferentes daquelas utilizadas em sistemas especialistas. Tenho dificuldade em ver como o reconhecimento de imagens, por exemplo, fará alguma diferença apreciável na prática de programação.
O mesmo é tentativa de reconhecimento de fala. O difícil na construção de software é decidir o que dizer, e não dizer. Nenhuma facilitação da expressão pode proporcionar mais do que ganhos marginais.

A tecnologia de sistemas especialistas, 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 para a construção de sistemas especialistas. Muitos cientistas de software estão trabalhando arduamente para aplicar essa tecnologia ao ambiente de construção de software.6 Qual é o conceito e quais são as perspectivas?

Um sistema especialista é um programa que contém um mecanismo de inferência generalizado e uma base de regras, projetado para receber dados de entrada e suposições e explorar as consequências lógicas através das inferências deriváveis da base de regras, produzindo conclusões e conselhos, e oferecendo-se para explicar seus resultados reconstituindo seu raciocínio para o usuário. Os mecanismos de inferência normalmente podem lidar com dados e regras difusos ou probabilísticos, além de lógica puramente determinística.

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

- A tecnologia do mecanismo de inferência é desenvolvida de forma independente da aplicação e depois 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 específicos da aplicação são codificadas na base de regras de maneira uniforme e são fornecidas ferramentas para desenvolver, alterar, testar e documentar a base de regras. 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 da complexidade da aplicação do próprio programa.

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

---

[5] Parnas, DL, “Aspectos de software de sistemas de defesa estratégicos”, *Communications of the ACM,* **28,** 12 (dezembro de 1985), pp. Também em *American Scientist,* **73,** 5 (setembro-outubro de 1985), pp.

[6] Balzer, R., “Uma perspectiva de 15 anos sobre programação automática”, em Mostow, op. cit.

Considere um consultor de testes imaginário, por exemplo. Na sua forma mais rudimentar, o sistema especialista de diagnóstico é muito parecido com uma lista de verificação de um piloto, oferecendo fundamentalmente sugestões sobre possíveis causas de dificuldade. À medida que a base de regras é desenvolvida, as sugestões tornam-se mais específicas, tendo em conta de forma mais sofisticada os sintomas de problemas relatados. Pode-se visualizar um assistente de depuração que oferece sugestões muito generalizadas a princípio, mas à medida que mais e mais estrutura do sistema é incorporada na base de regras, torna-se cada vez mais particular nas hipóteses geradas e nos testes que recomenda. Tal sistema especialista pode afastar-se radicalmente dos sistemas convencionais, pois sua base de regras provavelmente deveria ser modularizada hierarquicamente da mesma forma que o produto de software correspondente, de modo que, à medida que o produto é modificado modularmente, a base de regras de diagnóstico possa ser modificada modularmente como bem.

O trabalho necessário para gerar as regras de diagnóstico é o 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 adequadamente geral, com uma estrutura uniforme para regras e um bom mecanismo de inferência disponível, poderá na verdade reduzir o trabalho total de geração de casos de teste, bem como ajudar na manutenção ao longo da vida e nos testes de modificação. Da mesma forma, podemos postular outros conselheiros, provavelmente muitos deles e provavelmente simples para as outras partes da tarefa de construção de software.

Muitas dificuldades impedem a obtenção antecipada de consultores especializados úteis para o desenvolvedor do programa. Uma parte crucial do nosso cenário imaginário é o desenvolvimento de maneiras fáceis de passar da especificação da estrutura do programa à geração automática ou semiautomática de regras de diagnóstico. Ainda mais difícil e importante é a dupla tarefa da aquisição de conhecimento: encontrar especialistas articulados e autoanalíticos que saibam por que fazem as coisas; e desenvolver técnicas eficientes para extrair o que 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 ao serviço do programador inexperiente a experiência e a sabedoria acumulada dos melhores programadores. Esta não é uma contribuição pequena. A lacuna entre as melhores práticas de engenharia de software e a prática média é muito grande – talvez maior do que em qualquer outra disciplina de engenharia. Uma ferramenta que divulgue 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 esperassem que esta 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 ao que estava atualmente disponível para o programador.8*

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

---

[7] Mostow, op. cit.
[8] Parnas, 1985, op. cit.

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 uma biblioteca de métodos de solução e gerou os programas.

- Estas aplicações têm propriedades muito favoráveis: Os
- problemas são facilmente caracterizados por relativamente poucos parâmetros.
- Existem muitos métodos conhecidos de solução para fornecer uma biblioteca de alternativas.
- A análise extensiva levou a regras explícitas para a seleção de técnicas de solução, dados os parâmetros do problema.

É difícil ver como tais técnicas podem ser generalizadas para o mundo mais amplo do sistema de software comum, onde casos com propriedades tão claras são a exceção. É difícil até imaginar como este avanço na generalização poderia ocorrer.

Programação gráfica. Um assunto favorito 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 de chips VLSI, onde a computação gráfica desempenha um papel tão frutífero.
Às vezes, a abordagem é justificada pela consideração dos fluxogramas como o meio ideal para a concepção de programas e pelo fornecimento de recursos poderosos para sua construção.

Nada sequer convincente, muito menos excitante, emergiu ainda de tais esforços. EU 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, ele é melhor visto como a tentativa de Burks, von Neumann e Goldstine de fornecer uma linguagem de controle de alto nível desesperadamente necessária para seu computador proposto. Na forma lamentável, de múltiplas páginas e de caixa de conexão em que o fluxograma foi hoje elaborado, ele provou ser essencialmente inútil como uma ferramenta de design que os programadores desenham fluxogramas depois, e 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 detalhado e sério. A chamada “metáfora da área de trabalho” da estação de trabalho atual é, em vez disso, uma metáfora do “assento de avião”. Qualquer pessoa que tenha mexido um monte de papéis enquanto está sentado em uma carruagem entre dois passageiros corpulentos reconhecerá a diferença: só se pode ver muito poucas coisas ao mesmo tempo. A verdadeira área de trabalho fornece visão geral e acesso aleatório a várias páginas. Além disso, quando os ataques de criatividade são fortes, sabe-se que mais de um programador ou escritor abandonou o desktop por um 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, explosão de dados, estruturas de dados hierárquicas ou qualquer outra coisa, sentimos apenas uma dimensão do elefante de software intrinsecamente interligado. Se sobrepormos todos os diagramas gerados pelas muitas visões relevantes, será difícil extrair qualquer visão global. O VLSI

---

[9] Raeder, G., “Uma pesquisa das técnicas atuais de programação gráfica”, em RB Grafton e T. Ichikawa, eds., Special Issue on Visual Programming, Computer, 18, 8 *(agosto*

[10] de 1985), pp. 1995, op. cit., capítulo 15.

a analogia é fundamentalmente enganosa – um design de chip é um objeto bidimensional em camadas cuja geometria reflete sua essência. Um sistema de software não é.

Verificação do programa. Muito do esforço da programação moderna é direcionado para testes e reparos de bugs. Existe talvez uma solução mágica a ser encontrada eliminando os erros na fonte, na fase de design do sistema? Será que a produtividade e a confiabilidade do produto podem ser radicalmente melhoradas seguindo a estratégia profundamente diferente de provar que os projetos estão corretos antes que o imenso esforço seja investido em implementá-los e testá-los?

Não acredito que encontraremos magia aqui. A verificação do programa é um conceito muito poderoso e será muito importante para coisas como kernels seguros de sistemas operacionais.
A tecnologia não promete, porém, economizar 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. Também não há mágica aqui. As provas matemáticas também podem apresentar falhas. Portanto, embora a verificação possa reduzir a carga de testes 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 grande parte da essência da construção de um programa é, na verdade, a depuração da especificação.

Ambientes e ferramentas. Quanto mais ganhos podem ser esperados com a explosão de pesquisas em melhores ambientes de programação? A reação instintiva é que os problemas de grande recompensa foram os primeiros atacados e foram resolvidos: sistemas de arquivos hierárquicos, formatos de arquivos uniformes para ter interfaces de programas uniformes e ferramentas generalizadas. Editores inteligentes específicos de linguagem são desenvolvimentos ainda não amplamente utilizados na prática, mas o máximo que prometem é a ausência de erros sintáticos e erros semânticos simples.

Talvez o maior ganho ainda a ser alcançado no ambiente de programação seja o uso de sistemas de banco de dados integrados para acompanhar a miríade de detalhes que devem ser lembrados 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 dará alguns frutos tanto em termos de produtividade e confiabilidade. Mas, pela sua própria natureza, o retorno a partir de agora deverá ser marginal.

Estações de trabalho. Que ganhos podem ser esperados para a arte do software a partir do aumento certo e rápido na potência e na capacidade de memória da estação de trabalho individual? Bem, quantos MIPS alguém pode usar com sucesso? A composição e edição de programas e documentos são totalmente suportadas pelas velocidades atuais. A compilação poderia receber um impulso, mas um fator de 10 na velocidade da máquina certamente deixaria o tempo de reflexão como a atividade dominante na época do programador. Na verdade, parece ser assim agora.

Estações de trabalho mais poderosas certamente serão bem-vindas. Aprimoramentos mágicos deles não podemos esperar.

Ataques promissores à essência conceitual Embora nenhum avanço tecnológico prometa fornecer os 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 resultados constantes, embora nada espetaculares. progresso.

Todos os ataques tecnológicos aos acidentes do processo de software são fundamentalmente limitados pela equação da produtividade:

Hora da tarefa = ÿ *(Frequência) i x (Tempo) i*

Se, como acredito, os componentes conceptuais da tarefa estão agora a ocupar a maior parte do tempo, então nenhuma quantidade de actividade nos componentes da tarefa que são apenas a expressão dos conceitos pode proporcionar 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.

Comprar versus construir. A solução mais radical possível para a construção de 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 aplicações. Embora nós, engenheiros de software, tenhamos trabalhado na metodologia de produção, a revolução dos computadores pessoais criou não um, mas vários mercados de massa para software. Cada banca de jornal vendia revistas mensais que, classificadas por tipo de máquina, anunciavam e avaliavam dezenas de produtos a preços que variavam 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 pedágios e ambientes de software podem ser adquiridos imediatamente. Em outro lugar propus um mercado para módulos individuais.

Qualquer produto desse tipo é mais barato comprar do que construir de novo. Mesmo com um custo de US$ 100 mil, um software adquirido custa apenas cerca de um ano de programador.
E a entrega é imediata! Imediato, pelo menos para produtos que realmente existem, produtos cujo desenvolvedor pode encaminhar o cliente potencial para um usuário satisfeito. Além disso, esses produtos tendem a ser muito mais bem documentados e mantidos um pouco melhor do que software desenvolvido internamente.

O desenvolvimento do mercado de massa é, creio eu, a tendência de longo prazo mais profunda na engenharia de software. O custo do software sempre foi o custo de desenvolvimento, não o custo de replicação. Compartilhar esse custo mesmo entre alguns usuários reduz radicalmente o custo por usuário. Outra maneira de ver isso é que o uso de n cópias de um sistema de software multiplica efetivamente a produtividade de seus desenvolvedores por n. Isso é um aumento da produtividade da disciplina e da nação.

A questão principal, claro, é a aplicabilidade. Posso usar um pacote disponível no mercado para realizar 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. Durante a década de 1980, encontramos esses pacotes com alta demanda e uso generalizado. O que mudou?

Na verdade não são os pacotes. Eles podem ser um pouco mais generalizados e mais personalizáveis do que anteriormente, mas não muito. Na verdade, também não são os aplicativos. Se

qualquer coisa, as necessidades empresariais e científicas de hoje são mais diversas e mais complicadas do que as de há 20 anos.

A grande mudança ocorreu na relação custo de hardware/software. O comprador de um valor de US$ 2-A máquina de um milhão de dólares em 1960 sentiu que poderia pagar US$ 250 mil a mais por um programa de folha de pagamento personalizado, que penetrasse fácil e sem interrupções no ambiente social hostil aos computadores. Os compradores de máquinas de escritório de US$ 50 mil hoje não podem arcar com programas personalizados de folha de pagamento; então eles adaptam seus procedimentos de folha de pagamento aos pacotes disponíveis. Os computadores são agora tão comuns, se não tão amados, que as adaptações são aceitas como algo natural.

Há exceções dramáticas ao meu argumento de que a generalização dos pacotes de software mudou pouco ao longo dos anos: planilhas eletrônicas e sistemas simples de banco de dados. Estas ferramentas poderosas, tão óbvias em retrospectiva e, no entanto, de aparecimento tão tardio, prestam-se a uma miríade de utilizações, algumas bastante pouco ortodoxas. Agora há muitos artigos e até livros sobre como lidar com tarefas inesperadas com a planilha. Um grande número de aplicações que anteriormente teriam sido escritas como programas personalizados em Cobol ou Report Program Generator agora são feitas rotineiramente com essas ferramentas.

Muitos usuários agora operam seus próprios computadores todos os dias em aplicações variadas, sem nunca escrever um programa. Na verdade, muitos desses usuários não conseguem escrever novos programas para suas máquinas, mas mesmo assim são hábeis em resolver novos problemas com eles.

Acredito que a estratégia de produtividade de software mais poderosa para as organizações humanas hoje em dia é equipar os trabalhadores intelectuais ingénuos em informática na linha de fogo com computadores pessoais e bons programas generalizados de escrita, desenho, ficheiros e folhas de cálculo, e libertá-los. A mesma estratégia, com capacidades de programação simples, também funcionará para centenas de cientistas de laboratório.

Refinamento de requisitos e prototipagem rápida. A parte mais difícil da construção de 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 for feita de maneira errada. Nenhuma outra parte é mais difícil de corrigir mais tarde.

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. Geralmente não sabem quais perguntas devem ser respondidas e quase nunca pensaram no problema com os detalhes que devem ser especificados. Mesmo a resposta simples – “Fazer com que o novo sistema de software funcione como o nosso antigo sistema manual de processamento de informação” – é de facto demasiado 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, ao planejar qualquer atividade de software, é necessário permitir uma extensa iteração entre o cliente e o projetista como parte da definição do sistema.

Eu iria 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 terem construído e testado algumas versões do produto que eles estão especificando.

Portanto, um dos mais promissores 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 protótipo de sistema de software é aquele que simula as interfaces importantes e executa as funções principais do sistema pretendido, embora não esteja necessariamente vinculado às mesmas restrições de velocidade, tamanho ou custo de hardware. Os protótipos normalmente executam as tarefas principais da aplicação, mas não fazem nenhuma tentativa de tratar as exceções, responder corretamente a entradas inválidas, abortar de forma limpa, etc. O objetivo do protótipo é tornar real a estrutura conceitual especificada, para que o cliente possa testar para consistência e usabilidade Muitos dos procedimentos atuais de aquisição de software

baseiam-se na suposição de que é possível especificar antecipadamente um sistema satisfatório, obter propostas para sua construção, construí-lo e instalá-lo. Acredito que essa suposição está fundamentalmente errada e que muitos problemas de aquisição de software surgem dessa falácia. Portanto, eles não podem ser corrigidos sem uma revisão fundamental, que proporcione o desenvolvimento iterativo e a especificação de protótipos e produtos.

Desenvolvimento incremental – crescimento, não construção, software. Ainda me lembro do choque que senti em 1958, quando ouvi pela primeira vez um amigo falar sobre *construir* um programa, em vez de *escrevê* -lo. Num piscar de olhos, será ampliada toda a minha visão do processo de software. A mudança de 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 *andaimes.*

A metáfora do edifício perdeu a sua utilidade. É hora de mudar novamente. Se, como acredito, as estruturas conceptuais que construímos hoje são demasiado complicadas para serem especificadas antecipadamente com precisão, e demasiado complexas para serem construídas sem falhas, então devemos adoptar uma abordagem radicalmente diferente.

Voltemo-nos para a natureza e estudemos a complexidade dos seres vivos, em vez de apenas as obras mortas do homem. Aqui encontramos construções cujas complexidades nos emocionam. O cérebro por si só é intrincado além do mapeamento, poderoso além da imitação, rico em diversidade, autoprotetor e autorrenovador. O segredo é que se cresce, não se constrói.

O mesmo deve acontecer com nossos sistemas de software. Há alguns anos, Harlan Mills propôs que qualquer sistema de software deveria ser desenvolvido por meio de desenvolvimento incremental.11 Ou seja, primeiro o sistema deveria ser executado, mesmo que não faça nada de útil, exceto chamar o conjunto adequado de subprogramas fictícios. Então, pouco a pouco, ele é desenvolvido, com os subprogramas, por sua vez, sendo desenvolvidos em ações ou chamadas para stubs vazios no nível abaixo.

Tenho visto os resultados mais dramáticos desde que comecei a aplicar essa técnica aos construtores de projetos em minhas aulas de laboratório de engenharia de software. Nada na última década mudou tão radicalmente a minha própria prática, ou a sua eficácia. A abordagem requer um design de cima para baixo, pois é um crescimento de cima para baixo do software. Permite fácil retrocesso. Ele se presta aos primeiros protótipos. Cada função adicionada e nova provisão para dados ou circunstâncias mais complexas cresceram organicamente a partir do que já existe.

---

[11] Mills, HD, “Programação top-down em grandes sistemas”, *Técnicas de Depuração em Grandes Sistemas,* R. Rustin, ed., Englewood Cliffs, NJ, Prentice-Hall, 1971.

Os efeitos morais são surpreendentes. O entusiasmo aumenta 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 funcional. Acho que as equipes podem *desenvolver* entidades muito mais complexas em quatro meses do que conseguem *construir.*

Os mesmos benefícios podem ser obtidos em projetos grandes e pequenos.12

Grandes designers. A questão central de como melhorar a arte do software centra-se, como sempre, nas pessoas.

Podemos obter bons designs 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. Assim, um grande impulso nos Estados Unidos é promulgar boas práticas modernas. Novos currículos, nova literatura, novas organizações como o Instituto de Engenharia de Software, todos surgiram para elevar o nível da nossa prática de ruim para bom. Isto é totalmente apropriado.

No entanto, não acredito que possamos dar o próximo passo ascendente da mesma forma. Embora a diferença entre projetos conceituais ruins e bons possa residir na solidez do método de projeto, a diferença entre projetos bons e projetos excelentes certamente não reside. Grandes designs vêm de grandes designers. A construção de software é um processo criativo. Uma metodologia sólida pode capacitar e libertar a mente criativa; não pode inflamar ou inspirar o trabalho pesado.

As diferenças não são pequenas – é como Salieri e Mozart. Estudo após estudo mostra que os melhores projetistas produzem estruturas mais rápidas, menores, mais simples, mais limpas e produzidas com menos esforço. As diferenças entre o grande e o médio aproximam-se de uma ordem de grandeza.

Uma pequena retrospectiva mostra que, embora muitos sistemas de software excelentes e úteis tenham sido projetados por comitês e construídos por projetos de múltiplas partes, os sistemas de software que entusiasmaram fãs apaixonados são aqueles que são produtos de uma ou de algumas mentes projetistas, grandes designers. Considere Unix, APL, Pascal, Modula, a interface Smalltalk, até mesmo Fortran; e contraste com Cobol, PL/I, Algol, MVS/370 e MS-DOS (fig. 1)

Sim	Não
Unix	Cobol
APL	PL/1
Pascal	Algol
Módulo	MVS/370
Conversa fiada	MS DOS
Fortran	

Figura 1 Produtos interessantes

Assim, embora apoie fortemente os esforços de transferência de tecnologia e de desenvolvimento curricular actualmente em curso, penso que o esforço mais importante que podemos realizar é desenvolver formas de desenvolver grandes designers.

---

[12] Boehm, BW, “Um modelo espiral de desenvolvimento e aprimoramento de software”, *Computer,* **20,** 5 (maio de 1985), pp.

Nenhuma organização de software pode ignorar este desafio. Bons gestores, por mais escassos que sejam, não são mais escassos que bons designers. Grandes designers e grandes gestores são muito raros. A maioria das organizações despende um esforço considerável para encontrar e cultivar as perspectivas de gestão; Não conheço ninguém que se esforce tanto para encontrar e desenvolver os grandes designers dos quais dependerá, em última análise, a excelência técnica dos produtos.

Minha primeira proposta é que cada organização de software determine e proclame que os grandes designers são tão importantes para o seu sucesso quanto os grandes gerentes, e que se pode esperar que eles sejam igualmente nutridos e recompensados. Não só o salário, mas também os pré-requisitos de reconhecimento – tamanho do escritório, mobiliário, equipamento técnico pessoal, fundos para viagens, apoio do pessoal – devem ser totalmente equivalentes.

Como desenvolver grandes designers? O espaço não permite uma discussão longa, mas alguns passos são óbvios:

- Identifique sistematicamente os principais designers o mais cedo possível. Os melhores muitas vezes 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 designers de ponta, episódios de educação formal avançada e cursos de curta duração, todos intercalados com design individual e atribuições de liderança técnica.
- Ofereça oportunidades para que designers em crescimento interajam e estimulem uns aos outros. ÿ