Tradução de ‘No Silver Bullets’

Guilherme F. Batista (guilherme.da.franca.batista@gmail.com)

Guarulhos, 2017-02-20

SEM BALAS DE PRATA

Essência e acidentes de engenharia de software

Computer Magazine: 1987 de abril

por Frederick P. Brooks, Jr.,

University of North Carolina na Chapel Hill

Este artigo foi publicado pela primeira vez na Information Processing em 1986, ISBN No. 0444-7077-3, H. J. Kugler, Ed., Elsevia Science Publishers B.V. (North-holland) IFIP 1986. Digitalizado a partir de um fax de má qualidade copiado por Brad Cox, que é a razão para erros OCR ocasionais.

Eu (Brad Cox) escrevi diversos artigos (No Silver Bullet Revisited) e um livro (Superdistribution: Objects as Property on the Electronic Frontier) e fundei uma empresa (Superdistribution Inc.) baseada no desacordo com a conclusão principal deste artigo.

Resumidamente, Brooks argumenta que softwares nos desapontam porque são muito diferentes de bens tangíveis, sendo feitos de bits, não de átomos, e que as soluções que fazem os bens tangíveis irem ao encontro de nossas expectativas não podem, eventualmente, ser aplicadas. Este pressuposto é profundamente arraigado e só evidente no fato de que Brooks negligenciou sua menção como uma solução (por exemplo, como uma ‘bala de prata’) em seu artigo.

Eu defendo que os mercados robustos podem e vão recorrer ao software, uma vez que nós fornecemos os mecanismos descritos no livro e os mercados evoluíram por confiar neles. Isso colocaria o software na mesma curva de desempenho de preço que os bens tangíveis gozam hoje.

A criação de construções conceituais complexas é a essência; tarefas acidentais surgem representando as construções na linguagem. Os progressos anteriores reduziram assim as tarefas acidentais que o progresso futuro agora depende ao abordar a essência.

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

O projeto de software usual, pelo menos como visto por um gerente sem conhecimento técnico, tem algo desse personagem; ele é geralmente inocente e

correto, mas é capaz de se tornar um monstro de falta de horários, orçamentos explodidos e produtos defeituosos. Então nós ouvimos gritos desesperados por uma bala de prata – algo para fazer o custo do software cair tão rápido quanto o custo de hardware do computador.

Mas, conforme olhamos para o horizonte de uma década, não vemos nenhuma bala de prata. Não há um único desenvolvimento, quer na tecnologia, quer na técnica de gestão, que por si só promete um melhoramento expressivo na produtividade, na confiabilidade e na simplicidade. Neste artigo tentarei mostrar o porquê, examinando tanto a natureza do problema de software e as propriedades das balas propostas.

Ceticismo não é pessimismo, entretanto. Embora não vejamos descobertas surpreendentes – e, de fato, acredito que tal seja inconsistente com a natureza do software – muitas inovações encorajadoras estão em curso. Um esforço disciplinado e consistente para desenvolver, propagar e explorar essas inovações deve, de fato, produzir uma melhoria em ordem de grandeza. Não existe uma estrada real, mas há uma estrada.

O primeiro passo em direção ao controle de doenças foi a substituição das teorias sobre demônios e diversas outras pela teoria dos germes. Este mesmo passo, o começo da esperança, em si mesmo destruiu todas as esperanças de soluções mágicas. Isso mostrou aos trabalhadores que o progresso seria feito passo a passo, com grande esforço, e que um persistente e ininterrupto cuidado teria de ser pagos para uma disciplina de limpeza. Assim é a engenharia de software hoje.

TEM QUE SER DIFÍCIL? – DIFICULDADES ESSENCIAIS

Não só não existem balas de prata em vista, como a própria natureza do *software* torna improvável que haja alguma – nenhuma invenção fará para a produtividade, simplicidade e confiabilidade do *software* o que a eletrônica, transistores e integrações de larga escala fizeram para o *hardware* dos computadores.

Não podemos nunca esperar ver um ganho dobrado a cada dois anos.

Primeiramente, deve-se observar que a anomalia não é que o progresso do *software* é lento, mas que a evolução do *hardware* é rápida. Nenhuma outra tecnologia, desde o começo da civilização, tem visto ganho de preço de seis ordens de magnitude em 30 anos. Em nenhuma outra tecnologia pode-se optar por levar ganho tanto no desempenho quanto nos custos reduzidos. Estes ganhos decorrem da transformação da fabricação de computadores, de uma indústria de montagem para uma indústria de processo.

Em segundo lugar, para ver qual é a taxa de progresso que se pode esperar da tecnologia de software, vamos examinar as dificuldades dessa tecnologia. Seguindo Aristóteles, divido as dificuldades em essência, i.e., nas dificuldades inerentes na natureza do *software* e nos acidentes, ou seja, aquelas dificuldades que hoje afetam a sua produção, mas não são inerentes.

A essência da entidade do *software* é um constructo de conceitos interligados: conjuntos de dados, relações entre itens de dados, algoritmos e invocações de funções. Esta essência é abstrata na medida em que tal (um) constructo conceitual é o mesmo sob muitas representações diferentes. Não obstante, isto é altamente preciso e ricamente detalhado.

Acredito que a parte difícil na construção do *software* seja a especificação, o projeto e o teste dessa construção conceitual, não o trabalho de representá-lo e testar a

fidelidade da representação. Certamente ainda cometemos erros de sintaxe. Eles são leves comparado com os erros conceituais na maioria dos sistemas.

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

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

COMPLEXIDADE

Entidades de *software* são mais complexas por seu tamanho do que qualquer outro constructo humano, porque não há duas partes iguais (pelo menos acima do nível da declaração). Se elas forem, transformamos as duas partes similares em uma sub-rotina – aberta ou fechada. A este respeito, sistemas de *software* diferem profundamente de computadores, edifícios ou automóveis, onde abundam elementos repetitivos.

Computadores digitais são mais complexos do que a maioria das coisas que as pessoas constroem: Eles têm um número grande de estados. Isso faz com que conceber, descrever e testá-los seja muito difícil. Sistemas de *software* têm mais estados em ordem de grandeza do que os computadores.

Da mesma forma, o escalonamento (ampliação) de uma entidade de *software* não é meramente a repetição dos mesmos elementos em tamanhos maiores; é necessário um aumento no número de diferentes elementos. Na maioria dos casos, os elementos interagem entre si de forma não linear e a complexidade do todo aumenta mais do que linearmente.

A complexidade do *software* é uma propriedade essencial, não acidental. Assim, descrições de uma entidade de *software* que abstraem sua complexidade, geralmente abstraem sua essência. Durante três séculos, a matemática e as ciências físicas deram grandes passos construindo modelos simplificados de fenômenos, derivando propriedades dos modelos e verificando as mesmas por experiência. Este paradigma funcionou porque as complexidades ignoradas nos modelos não eram as propriedades essenciais dos fenômenos. Isso não funciona quando as complexidades são a essência.

Muitos dos problemas clássicos de desenvolvimento de produtos de *software* derivam dessa complexidade essencial e seus aumentos não lineares com o tamanho. Da complexidade vem a dificuldade de comunicação entre os membros da equipe, o que leva a falhas no produto, excesso de custos e atrasos no cronograma. Da complexidade vem a dificuldade de enumerar, menos entendimento, todos os estados possíveis do programa e daí vem a falta de confiabilidade. Da complexidade da função vem a dificuldade de invocá-la, o que torna os programas difíceis de usar. Da complexidade da estrutura vem a dificuldade de estender programas a novas funcionalidades sem criar efeitos colaterais. Da complexidade da estrutura vêm os estados não visualizados, que constituem vulnerabilidades de segurança.

Não só problemas técnicos, mas problemas de gerenciamento vêm da complexidade. Isso torna a visão geral difícil, impedindo assim a integridade conceitual. Faz com que seja difícil encontrar e controlar todas as pontas soltas. Isso cria um fardo tremendo de aprendizagem e compreensão que faz a rotatividade de pessoal um desastre.

CONFORMIDADE

Os profissionais do *software* não são os únicos a enfrentar a complexidade. A física lida com objetos terrivelmente complexos, mesmo no nível fundamental. Contudo, o

físico trabalha em uma fé firme de que existem princípios unificadores a serem encontrados, seja nos *quarks* ou na teoria do campo unificado. Einstein argumentou que deve haver simplificação da natureza, porque Deus não é temperamental ou arbitrário.

Não existe tal fé que conforta o engenheiro de *software*. Grande parte da complexidade que ele deve dominar é a complexidade arbitrária, forçada sem razão pelas muitas instituições humanas e sistemas aos quais suas interfaces devem estar em conformidade. Estas diferem de interface para interface, e de vez em quando, não por necessidade, mas apenas porque foram projetados por pessoas diferentes e não por Deus.

Em muitos casos, o *software* deve conformar-se porque é o lançamento mais recente em cena. Em outros, deve conformar-se porque é percebido como o mais adaptável. Mas, em todos os casos, muita complexidade vem da conformidade para outras interfaces; esta complexidade não pode ser simplificada por qualquer redesenho do software.

MUTABILIDADE

A entidade de *software* está constantemente sujeita a pressões para mudança. Naturalmente, assim são os edifícios, carros, computadores. Mas coisas manufaturadas são raramente mudadas após a manufatura. Elas são mudadas por modelos posteriores, ou mudanças essenciais são incorporadas em cópias do mesmo projeto básico. *Callbacks* de automóveis estão bem menos frequentes hoje em dia. Mudanças de campo em computadores, um pouco menos. Ambos são muito menos frequentes do que modificações de *softwares* 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 de mudança. Em parte, é porque o *software* pode ser mudado facilmente – é pura coisa do pensamento, infinitamente maleável. Os edifícios, na verdade, mudam, mas os altos custos de mudança, entendidos por todos, servem para amortecer os caprichos das mudanças.

Todo *software* de sucesso sofre alteração. Dois processos estão em curso. Primeiro, como um produto de *software* é encontrado para ser útil, pessoas o testam em novos casos no limite ou além de seu 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 sobrevive além da vida normal da máquina para a qual foi escrito pela primeira vez. Se não para novos computadores, pelo menos para novos discos, novos displays, novas impressoras. E o *software* deve conformar-se com esses novos veículos de oportunidade.

Em suma, o produto de *software* está embutido em uma raiz cultural de aplicações, usuários, leis e máquinas. Todos estes mudam continuamente e suas mudanças forçam, inexoravelmente, uma mudança no produto de *software*.

INVISIBILIDADE

O *software* é invisível e impossível de visualizar. Abstrações geométricas são ferramentas poderosas. A planta de um edifício ajuda o arquiteto e o cliente a avaliar espaços, fluxos de tráfego, pontos de vista. Contradições e omissões tornam-se óbvias.

Apesar do progresso nas estruturas de *software* de re*ic'dng e dmPlifYinB (palavras indecifráveis), elas permanecem intrinsecamente invisíveis e não permitem que a mente use algumas de suas ferramentas conceituais mais poderosas.

Desenhos a escala de peças mecânicas e modelos de figuras lineares 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. Assim, não tem nenhuma representação geométrica acabada da mesma forma que a terra tem mapas, *chips* de silício têm diagramas e computadores têm esquemas de conectividade. Logo que tentamos diagramar a estrutura do *software*, descobrimos que ela constitui não um, mas vários gráficos gerais direcionados e sobrepostos um sobre o outro. Os vários gráficos podem representar o fluxo de controle, o fluxo de dados, padrões de dependência, sequência de tempo e relações de *namespace*. Esses gráficos geralmente não são nem planares e muito menos hierárquicos. De fato, uma das maneiras de estabelecer o controle conceitual sobre tal estrutura é forçar o corte de ligação até que um ou mais dos gráficos se tornem hierárquicos.

Apesar do progresso em restringir e simplificar as estruturas do *software*, eles permanecem intrinsecamente invisíveis e, portanto, não permitem que a mente use algumas de suas ferramentas conceituais mais poderosas. Esta carência não só impede o progresso de concepção dentro de uma mente, como dificulta a comunicação entre diferentes outras.

DESCOBERTAS PASSADAS ENVOLVERAM DIFICULDADES ACIDENTAIS

Se examinarmos os três passos no desenvolvimento de tecnologia de *software* que foram mais frutíferos no passado, descobrimos que cada um deles atacou uma dificuldade diferente na construção do *software*, mas que essas dificuldades foram acidentais, não essenciais. Podemos também ver os limites naturais para a extrapolação de cada um desses ataques.

LINGUAGENS DE ALTO NÍVEL

Certamente, o golpe mais poderoso para a produtividade, confiabilidade e simplicidade do *software*, tem sido o uso progressivo de linguagens de alto nível para programação. A maioria dos observadores credita esse desenvolvimento com pelo menos um quinto de ganho na produtividade, e com ganhos concomitantes em confiabilidade, simplicidade e compreensão.

O que uma linguagem de alto nível conquista? Ela livra o programa de muitas das complexidades acidentais. Um programa abstrato consiste em construções conceituais: operações, tipos de dados, sequências e comunicação. O verdadeiro programa da máquina está preocupado com *bits*, registros, condições, canais, discos e outros. Na medida em que a linguagem de alto nível incorpora as construções que se deseja no programa abstrato e evita todas as construções inferiores, 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. Certamente, o nível de nosso pensamento sobre estruturas de dados, tipos de dados e operações está aumentando constantemente, mas em uma taxa cada vez menor. E o desenvolvimento da linguagem aproxima-se cada vez mais da sofisticação dos usuários.

Além disso, em algum momento, a elaboração de uma linguagem de alto nível cria um peso de erudição de ferramentas que aumenta, não diminui, a tarefa intelectual do usuário que raramente usa as construções esotéricas.

COMPARTILHAMENTO DE TEMPO

O *time-sharing* trouxe uma grande melhoria na produtividade dos programadores e na qualidade de seus produtos, embora não tão grande quanto as trazidas pelas linguagens de alto nível.

O *time-sharing* ataca uma dificuldade bem diferente. Ele preserva a urgência e, portanto, permite manter uma visão geral da complexidade. O lento tempo gasto da programação em lotes significa que, inevitavelmente se esquece das minúcias, senão da própria ideia, do que se pensava quando parou de programar e pediu pela compilação e execução. Esta interrupção é onerosa em tempo pois é preciso atualizar a memória. O efeito mais grave pode muito bem ser o decaimento da compreensão de tudo o que está acontecendo em um sistema complexo. ’

Um contratempo vagaroso, como as complexidades da linguagem de máquina, é uma dificuldade acidental e não essencial do processo do *software*. Os limites da contribuição potencial de *time-sharing* derivam diretamente disso. O efeito principal do *time-sharing* é encurtar o tempo de resposta do sistema. Como este tempo de resposta vai para zero, ele passa, em algum ponto, o limiar humano de notoriedade, cerca de 100 milissegundos. Para além desse limite, benefícios não podem ser esperados.

AMBIENTES DE DESENVOLVIMENTO DE PROGRAMAS INTEGRADOS

Unix e Interslip, os primeiros ambientes de programação integrados a entrar em uso geral, parecem ter melhorado a produtividade por fatores integrais. Por quê?

Eles atacam as dificuldades acidentais que resultam do uso do uso de programas específicos de modo agregado, fornecendo bibliotecas integradas, formatos de arquivos unificados, pipes e filters. Como resultado, as estruturas conceituais que, a princípio, poderiam sempre chamar, alimentar e usar uns aos outros, pode facilmente fazê-lo na prática.

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

Por causa desses sucessos, os ambientes são o tema de grande parte da pesquisa de engenharia de *software* de hoje. Nós olharemos para suas promessas e limitações num futuro breve.

ESPERANÇAS PARA A PRATA

Agora vamos considerar os desenvolvimentos técnicos que são mais frequentemente avançados como potenciais balas de prata. Eles tratam de quais problemas-- os problemas de essência ou as dificuldades acidentais remanescentes? Eles oferecem avanços revolucionários ou incrementais?

Um desses desenvolvimentos é a ADA, uma linguagem de alto nível de uso geral da década de 1980. Ada não só reflete melhorias evolutivas nos conceitos de linguagem, mas de fato incorpora recursos para incentivar a projeção e modularização modernas. Talvez a filosofia de Ada seja mais avançada que sua própria linguagem, pois é a filosofia da modularização, dos tipos abstratos de dados, da estruturação hierárquica. Ada é muito rica, um resultado natural do processo pelo qual os requisitos foram estabelecidos em seu projeto. Isso não é fatal, pois os subconjuntos de vocabulário de trabalho podem resolver o problema de aprendizado e os avanços de *hardware* nos darão MIPS (milhões de instruções por segundo) baratos para pagar os custos de compilação. Aumentar a estruturação de sistema de *software* é realmente um uso bom para o aumento de MIPS que nossos dólares vão pagar. Os

sistemas operacionais, altamente criticados na década de 1960 por seus custos de memória e ciclos, provaram ser uma excelente forma de usar alguns dos MIPS e bytes de memória baratos da onda passada de hardware[1].

No entanto, Ada não irá revelar-se como sendo a bala de prata que mata o monstro de produtividade de *software*. Afinal de contas, ela é apenas mais uma linguagem de alto nível e a maior recompensa dessas linguagens veio da primeira transição – a transição das complexidades acidentais da máquina para a declaração mais abstrata de soluções passo-a-passo. Uma vez que os acidentes foram removidos, os restantes serão menores e a recompensa de sua remoção será certamente menor.

Prevejo que daqui a uma década, quando a eficácia de Ada for avaliada, ver-se-á ter feito uma diferença substancial, mas não por causa de qualquer recurso particular 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 seu uso ocasionou na formação de programadores em técnicas modernas de design de software.

PROGRAMAÇÃO ORIENTADA A OBJETOS

Muitos estudantes da arte esperam com mais esperança pela programação orientada a objetos do que qualquer outro modismo técnico do dia[2]. Eu estou entre eles. Mark Shernman de Dartmouth aponta no CSNet News que é preciso ter cuidado ao distinguir duas ideias separadas que vão sob este nome: tipos de dados abstratos e tipos hierárquicos. O conceito de tipo de dado abstrato é que o tipo de objeto deve ser definido pelo nome, um conjunto de valores apropriados e um conjunto de operações apropriadas em vez da sua estrutura de armazenamento, que deve ser oculta. Exemplos são os pacotes de Ada (com tipos privados) e os módulos do Modula.

Tipos hierárquicos, como as classes do Simula-7, permitem definir interfaces gerais que podem refinadas fornecendo tipos subordinados. Os dois conceitos são ortogonais –um pode ter hierarquias sem ocultar e o outro ocultar sem hierarquias. Ambos os conceitos representam avanços reais na arte da construção de software.

Cada um deles ainda remove uma outra dificuldade acidental do processo, permitindo que o projetista expresse a essência do projeto sem ter que expressar grandes quantidades de material sintático que não adiciona conteúdo informativo. Para ambos os tipos, abstratos e hierárquicos, o resultado é remover um tipo de dificuldade acidental de ordem superior e permitir uma expressão de projeto de ordem superior.

No entanto, tais avanços não podem fazer mais do que remover todas as dificuldades acidentais da expressão do projeto. A complexidade do projeto em si é essencial, e tais ataques não fazem qualquer mudança nela. Um ganho em ordem de grandeza pode ser feito pela programação orientada a objetos, somente se a desnecessária especificação de tipo, ainda em nossa linguagem de programação, for nove décimos do trabalho envolvido na concepção de um programa. Eu duvido disso.

INTELIGÊNCIA ARTIFICIAL

Muitas pessoas esperam avanços na inteligência artificial para fornecer o avanço revolucionário que dará ganhos em ordem de grandeza na produtividade e qualidade do software[3]. Eu não. Para ver por que, devemos dissecar o que se entende por inteligência artificial.

D. L. Parnas esclareceu o caos terminológico[4]: Duas definições bastante

diferentes de I.A estão em uso comum hoje em dia.

I.A.-1: O uso de computadores para resolver problemas que anteriormente só poderiam ser resolvidos pela aplicação da inteligência humana.

I.A.-2: O uso de um conjunto específico de técnicas de programação conhecido como heurística ou programação baseada em regras. Nesta abordagem, especialistas humanos são estudados para determinar quais heurísticas ou regras básicas eles usam na resolução de problemas... O programa é projetado para resolver um problema da mesma maneira que os seres humanos parecem resolvê-lo.

A primeira definição tem um significado ambíguo – algo pode caber na definição I.A-1 hoje, mas, uma vez que vemos como o programa funciona e entendemos o problema, não mais pensaremos nele como I.A… Infelizmente, não consigo identificar um corpo de tecnologia que é exclusivo para este campo… A maior parte do trabalho é específica do problema, e alguma abstração ou criatividade é necessária para ver como transferi-la.

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 usadas em sistemas especialistas. Tenho dificuldade em ver como o reconhecimento de imagens, por exemplo, fará qualquer diferença apreciável na prática de programação. O mesmo problema é válido no reconhecimento de fala. A coisa difícil sobre a construção de *software* é decidir o que se quer dizer, não o dizer. Nenhuma facilitação na expressão pode dar mais do que ganhos insignificantes.

A tecnologia de sistemas especialistas, A.I.-2, merece uma seção própria.

SISTEMAS ESPECIALISTAS

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 duro aplicando essa tecnologia para o ambiente de construção de *software*[[3]-[5]]. Qual é o conceito e quais são as perspectivas?

Um sistema especialista é um programa que contém uma estrutura de inferência geral e uma base de regras, recebe dados de entrada e pressupostos, explora as inferências deriváveis da base de regras, produz conclusões e proposições, e as oferece para explicar seus resultados rastreando seu raciocínio para o usuário. ' A estrutura, ou mecanismo de inferência, pode, tipicamente, lidar com dados e regras difusas ou probabilísticas, além da lógica puramente determinística.

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

A tecnologia do mecanismo de inferência é desenvolvida de uma maneira independente de aplicação e, em seguida, aplicada a diversos usos. Pode-se justificar muito esforço nas estruturas de inferência. Na verdade, essa tecnologia está bem avançada.

* As partes mutáveis dos materiais peculiares da aplicação são codificados na base de regras de forma uniforme, e ferramentas são fornecidas para desenvolver, alterar, mudar, testar e documentar a base de regras. Isso regulariza muito da complexidade da aplicação.

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 por essa tecnologia é a separação da complexidade da aplicação do programa em si.

Como essa tecnologia pode ser aplicada à tarefa de engenharia de *software*? De muitas maneiras: tais sistemas podem sugerir regras de interface, aconselhar sobre estratégias de teste, lembrar a frequência dos tipos de erros, e oferecer outras dicas de otimização.

Considere um conselheiro de testes imaginário, por exemplo. Em sua forma mais rudimentar, o sistema de diagnóstico especialista é muito parecido com a lista de controle de um piloto, apenas enumerando sugestões sobre as possíveis causas de dificuldade. À medida que a estrutura do sistema é mais e mais incorporada na base de regras, e à medida que a base de regras assume um repositório mais sofisticado dos sintomas de problemas relatados, o conselheiro de testes se torna cada vez mais particular nas hipóteses que gere e nos testes que recomenda. Tal sistema especialista pode afastar-se mais radicalmente dos sistemas convencionais na medida em que sua base de regras deve ser hierarquicamente modularizada da mesma forma que o produto de *software* corresponde, de modo que, à medida que o produto é modificado modularmente, a base de regras de diagnóstico pode ser modificada modularmente também.

O trabalho necessário para declarar as regras de diagnóstico teria de ser feito, de qualquer maneira, na geração do conjunto de casos de teste para os módulos e para o sistema. Se for feito de maneira bastante geral, com uma estrutura uniforme para as regras e uma boa máquina diferencial disponível, pode realmente reduzir o trabalho de geração de casos de teste e bdp[indecifrável], bem como a manutenção vitalícia e testes de modificação. Da mesma forma, pode-se postular outros conselheiros provavelmente muitos e simples, para as outras partes da tarefa de construção do *software*.

Muitas dificuldades ficam no caminho da realização adiantada de conselheiros úteis para o desenvolvedor do sistema. Uma parte crucial de nosso cenário imaginário é o desenvolvimento de maneiras fáceis de passar da especificação da estrutura do programa para a 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 autoanalíticos que saibam por que fazem coisas e desenvolver técnicas eficientes para extrair o que sabem e destilar isso em bases de regras. O pré-requisito essencial para a construção de um sistema especialista é ter um especialista.

A contribuição mais poderosa dos sistemas especialistas será, certamente, colocar à disposição do programador inexperiente a experiência e sabedoria acumulada dos melhores programadores. Esta não é uma pequena contribuição. A diferença entre a melhor prática de engenharia de *software* e a prática média é muito ampla – talvez mais ampla do que qualquer outra disciplina de engenharia. Uma ferramenta que divulgue boas práticas seria importante.

PROGRAMAÇÃO AUTOMÁTICA

Por quase 40 anos, pessoas têm se antecipado e escrito sobre “programação automática”, ou a geração de um programa para resolver um problema a partir da declaração das especificações do problema. Hoje alguns escrevem como se esperassem que essa tecnologia proporcionasse o próximo grande avanço.

Parnas propõe que o termo é utilizado por glamour, não por conteúdo semântico.

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

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

Pode-se encontrar exceções. A técnica de construção de geradores (geradores de artefatos) é muito poderosa e é rotineiramente usada para uma boa vantagem em programas de classificação. Alguns sistemas para integrar equações diferenciais também permitiram a especificação direta do problema, e os sistemas avaliaram os parâmetros, escolhidos a partir de uma biblioteca de métodos de solução, e geraram os programas.

Estas aplicações têm propriedades bastante favoráveis:

- Os problemas são facilmente caracterizados por relativamente poucos parâmetros.

- Existem muitos métodos conhecidos de solução para proporcionar 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 generalizam para o mundo mais amplo do sistema de *software* comum, onde casos com tais propriedades puras são as exceções. É difícil mesmo imaginar como esse avanço na generalização poderia ocorrer.

PROGRAMAÇÃO GRÁFICA

Um assunto predileto para dissertações de doutorado em engenharia de *software* é computação gráfica, ou visual, a aplicação de computação gráfica para o desenvolvimento de *software*[[6][7]]. Às vezes, a promessa oferecida por tal abordagem é postulada por analogia com os *chips* VLSI, nos quais a computação gráfica desempenha um papel frutífero. Às vezes o teórico justifica a abordagem considerando os fluxogramas como o meio de projeto ideal e fornecendo poderosos recursos para construí-los.

Nada convincente, muito menos emocionante, ainda emergiu de tais esforços. Estou persuadido de que nada vai acontecer.

Primeiramente, como já discuti em outro lugar[8], o fluxograma é uma abstração muito pobre da estrutura de software. Na verdade, isso é mais visualizável como Burks, von Neumann e Goldstine tentaram desesperadamente fornecer uma linguagem de controle de alto nível para o seu computador proposto. O fluxograma, do modo que foi elaborado, lamentável, com múltiplas páginas e com formato de conexão em caixas, provou ser inútil como projeto.

O desenvolvimento do mercado de massa é, creio eu, a mais profunda tendência de longo prazo na engenharia de *software*. O custo do *software* sempre foi o custo de desenvolvimento, não o custo de replicação. Compartilhar esse custo, até mesmo entre alguns usuários, reduz radicalmente o custo por usuário. Outra forma de olhar para 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 de disciplina e de pessoal.

A questão chave, naturalmente, é a aplicabilidade. Posso usar um pacote pronto para uso (existente) para executar minha tarefa? Uma coisa surpreendente aconteceu aqui. Durante as décadas de 1950 e 1960, estudos após estudos mostraram que os usuários não usariam pacotes de produtos prontos de folha de pagamento, controle de estoque, contas a receber e assim por

diante. Os requisitos eram bastante específicos, a variação caso a caso era muito grande. Durante a década de 1980, encontramos esses pacotes em alta demanda e uso generalizado. O que mudou?

Não foram os pacotes. Eles podem ser um pouco mais gerais e customizáveis que antigamente, mas não muito. Nem mesmo as aplicações. Se alguma coisa mudou, foram as necessidades de negócios e científicas que hoje são mais diversificadas e complicadas do que as de 20 anos atrás.

A grande mudança tem sido a relação de custo de *hardware/software*. Em 1960, o comprador de uma máquina de dois milhões de dólares sentiu que poderia $ 250,000 a mais por um programa personalizado de folha de pagamento. Hoje, o comprador de uma máquina de U$ 50.000,00 não pode pensar em ter recursos para um programa personalizado de folha de pagamento, assim ele adapta o procedimento da folha de pagamento aos pacotes disponíveis. Computadores são agora tão comuns, senão amados, que as adaptações são aceitas como uma questão de rotina.

Existem exceções dramáticas * argumento de que a geração de pacotes de *software* mudou pouco ao longo dos anos: planilhas eletrônicas e sistemas simples de banco de dados. Estas poderosas ferramentas, óbvias em retrospecto e tão tardias em aparecer, se prestam a inúmeros usos, alguns bastante heterodoxos.

Agora, artigos e até mesmo livros abundam sobre como lidar com tarefas inesperadas com a planilha. Um grande número de aplicações que formalmente teriam sido escritos como programas personalizados em Cobol ou no Report Program Generator, agora são facilmente feitos com essas ferramentas.

Muitos usuários agora operam seus computadores, dia após dia, em várias aplicações sem nunca terem escrito um programa. De fato, muitos desses usuários não podem escrever novos programas para suas máquinas, mas eles são, no entanto, capazes de resolver novos problemas com eles.

Acredito que a estratégia de produtividade de *software* mais poderosa para muitas organizações hoje seja equipar os trabalhadores intelectuais ingênuos do computador, que estão na linha de frente, com computadores pessoais e programas gerais de escrita, desenho, arquivo e planilha e depois deixá-los livres. A mesma estratégia, realizada com pacotes gerais de matemática e estatística e algumas competências básicas de programação, também funcionará para centenas de cientistas de laboratório.

REFINAMENTO DE REQUISITOS E PROTOTIPAGEM RÁPIDA

A parte mais difícil na 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 com as pessoas, com as máquinas e com os outros sistemas de *software*. Nenhuma outra parte do trabalho prejudica tanto o sistema resultante se for feito de forma errada. Nenhuma outra parte é mais difícil de retificar posteriormente.

Portanto, a função mais importante que o construtor de *software* executa para o cliente é a extração iterativa e o refinamento dos requisitos do produto. A verdade é que o cliente não sabe o que quer. O cliente geralmente não sabe quais perguntas devem ser respondidas, e ele quase nunca pensa nos detalhes do problema para a especificação. Até mesmo a simples resposta - “Faça o novo sistema funcionar como nosso antigo sistema manual de processamento de informações” - é muito simples. Nunca se quer exatamente isso.

Sistemas de *software* são, além disso, coisas que agem, que se movem, que funcionam. É difícil imaginar a dinâmica dessa ação. Portanto, no planejamento de qualquer atividade de projeto de *software*, é necessário permitir uma iteração extensiva entre o cliente e o projetista, como parte da definição do sistema.

Eu iria mais longe e afirmaria que é realmente impossível para um cliente, mesmo trabalhando com um engenheiro de *software*, especificar completa e precisamente os requisitos exatos de um produto de *software* moderno, sem antes provar algumas versões do produto.

Portanto, um dos mais promissores esforços tecnológicos atuais, e que ataca a essência, não os acidentes, do problema de *software*, é o desenvolvimento de abordagens e ferramentas para a prototipagem rápida de sistemas, pois ela faz parte da especificação iterativa de requisitos.

Um protótipo de sistema de *software* é aquele que simula as interfaces importantes e executa as principais funções do sistema pretendido, embora não esteja necessariamente ligado à mesma velocidade de *hardware*, tamanho ou restrições de custo. Protótipos normalmente executam as tarefas principais da aplicação, mas não tentam lidar com tarefas excepcionais, responder corretamente a entradas inválidas ou encerrar de forma correta. O objetivo do protótipo é tornar real a estrutura conceitual especificada para que o cliente possa testá-lo em termos de consistência e usabilidade.

Muito do processo atual de aquisição de *software*, baseia-se na suposição de que se pode especificar antecipadamente um sistema satisfatório, obter ofertas para sua construção, tê-lo construído e instalá-lo. Eu acho que essa suposição é fundamentalmente errada, e ela traz muitos problemas de aquisição de *software*. [[9] [10]] [11][12]

AGRADECIMENTOS

Agradeço a Gordon Bell, Bruce Buchanan, Rick Hayes-Roth, Robert Patrick e, sobretudo, a David Parnas pelas suas ideias estimulantes e a Rebekah Bierly pela produção técnica deste artigo.

Frederick P. Brooks é professor de Ciência da Computação na University of North Carolina em Chapel Hill. Ele é mais conhecido como o “pai da família de computadores IBM System/360”, tendo atuado como gerente de projeto para o hardware da System/360 e mais tarde como gerente de projeto para o software da Operating System/360.

Em Chapel Hill, Brooks fundou o UNC Department of Computer Science e participou da fundação e administração da Microelectronics Center of North Carolina, do Triangle Universities Computation Center e do North Carolina Educational Computing Service. Ele recebeu a National Medal of Technology, a Guggenheim fellowship e o prêmio Computer Pioneer da Computer Society of the IEEE.

Brooks recebeu seu doutorado de Harvard, onde foi aluno de Howard Aiken.

Leitores podem escrever para F.P. Brooks na University of North Carolina, Department of Computer Science, Chapel Hill, NC 27514.

PARA MATAR O LOBISOMEN

Por que uma bala de prata? Magia, é claro. A prata é identificada com a Lua e, portanto, possui propriedades mágicas. Uma bala de prata oferece a maneira mais rápida, poderosa e segura de matar o mais rápido, poderoso e incrivelmente perigoso lobisomem. E o que poderia ser mais natural que usar a ‘lua de metal’ para destruir uma criatura transformada sob a luz da Lua cheia?

A lenda do lobisomem é, provavelmente, uma das mais antigas lendas de monstros. Heródoto, no século V a.C, nos deu o primeiro relato escrito de lobisomens quando mencionou uma tribo, ao norte do Mar Negro, chamada Neuri que, supostamente, transformavam-se em lobos alguns dias do ano. Heródoto escreveu que ele não acreditava nisso.

Com exceção dos céticos, muitos acreditavam em pessoas se transformando em lobos ou outros animais. Na Europa medieval, muitas pessoas foram mortas, pois pensavam que elas fossem lobisomens. Naquele tempo, não era preciso ser mordido por um lobisomem para se tornar um. Um pacto com o diabo, usar uma poção especial, usar um cinto especial ou ser amaldiçoado por uma bruxa, tudo isso poderia transformar uma pessoa em lobisomem. No entanto, os lobisomens medievais podiam ser feridos e mortos por armas normais. O problema era superar sua astúcia e furtividade.

Passemos ao lobisomem fictício, não lendário. O primeiro grande filme de lobisomem, *The Werewolf of London*, de 1935, criou o homem-lobo de duas pernas que se transformou em um monstro quando a lua estava cheia. Ele se tornou um lobisomem depois de ser mordido por outro, e só poderia ser morto com uma bala de prata. Parece familiar?

Na verdade, devemos muitas das ideias de hoje sobre lobisomens a Lon Chaney Jr em sua inesquecível versão de *Wolf Man*, de 1941. Os filmes subsequentes raramente se afastavam da mitologia mostrada nesse filme. No entanto, o filme de Chaney se afastou da mitologia original do lobisomem.

Você acreditaria que antes da ficção se tornar lenda, lobisomens não eram afetados por balas de prata? Eram os vampiros que não poderiam aguentar a prata. Mas é claro, se você confiar nas lendas, sua única salvação, se estiver desarmado e sofrendo um ataque de lobisomem, é subir num freixo ou correr para dentro de um campo de centeio. Não tão fáceis de se encontrar num cenário urbano, e dificilmente reconhecidos pelo público médio do cinema.

Para o que você deve estar atento? Pessoas cujas sobrancelhas crescem juntas, cujo dedo indicador é maior que o dedo médio e que tem pelos nas palmas das mãos. Dentes vermelhos ou pretos são um sinal definitivo de possíveis problemas. Mas tome cuidado. Os mesmos sintomas marcam as pessoas que sofrem de hipertricose (pessoas nascidas com pelos cobrindo seus corpos) ou porfiria. No caso da porfiria, o corpo de uma pessoa produz toxinas chamadas porfirinas. Consequentemente, a luz torna-se dolorosa, a pele é coberta por pelos e os dentes podem ficar vermelhos. Para piorar a reputação da vítima, seu comportamento cada vez mais bizarro faz com que as pessoas suspeitem ainda mais dos outros sintomas. Parece bastante provável que os contagiados pela doença contribuíram, involuntariamente, para a formação da lenda atual, embora em tempos passados não fossem acusados de ter tendências assassinas.

Vale a pena notar que a tradição do cinema, muitas vezes, faz do lobisomem um personagem bastante simpático, um inocente transformado em monstro. Como disse o cigano em *The Wolf Man*,

Mesmo um homem que é puro de coração,

E diz suas orações à noite,

Pode se tornar um lobo quando o acônito floresce,

E a Lua está cheia e brilhante.

REFERÊNCIAS

[1] PARNAS, D. L. "Designing Software for Ease of Extension and Contraction". IEEE Trans. **Software Engineering**, v. 5, n. 2, p. 128-138, Mar 1979.

[2] BOOCH, G. "Object-oriented Design". **Software Engineering with Ada**, 1983, Benjamin/Cummings, Menlo Park, Calif.

[3] IEEE Trans. **Software Engineering** (special issue on artificial intelligence and software engineering), J. Mostow, guest ed., v. 11, n. 11, Nov 1985.

[4] PARNAS, D. L. "Software Aspects of Strategic Defense Systems". **American Scientist**, Nov. 1985.

[5] BAKER, R. "A 15-Year Perspective on Automatic Programming," IEEE Trans. **Software Engineering**

(special issue on artificial intelligence and software engineering), *. Mostow, guest ed., v. 11, n. 11, p. 1257-1267, Nov 1985.

[6] GRAPHTON, R. B.; ICHIKAWA, T. **Computer** (special issue on visual programming). guest eds., v. 18, n. 8, Aug. 1985.

[7] RAEDER, G. "A Survey of Current Graphical Programming Techniques," **Computer** (special issue on

visual programming), R.B. Graphton and T. Ichikawa, guest eds., v.18, n. 8, p.11-25, Aug. 1985.

[8] BROOKS, F. P. The Mythical Man-Month, 1975, Addison-Wesley. **Reading.** Mass. New York:

Chapter 14.

[9] Defense Science Board. Report of the Task Force on Military Software. In press.

[10] MILLS, H. D. "Top-Down Programming in Large Systems" in Debugging Techniques in Large

Systems, R. Ruskin, ed. Prentice-Hall, **Englewood Cliffs**. N.*., 1971.

[11] BOEHM, B. W. "A Spiral Model of Software Development and Enhancement," 1985, **TRW tech.**

report 21-371-85, TRW, Inc., I Space Park. Redondo Beach, CA 90278.

[12] SACKMAN, H.; ERIKSON, W. J.; GRANT, E. E. "Exploratory Experimental Studies Comparing Online and Offline Programming Performance." **CACM**, v. II, n. I, p. 3-11, Jan. 1968.