Universidade da Região de Joinville
Bacharelado em Engenharia de Software
Fundamentos de Engenharia de Software

Uma breve História da Engenharia de Software

Niklaus Wirth

Niklaus Emil Wirth (Winterthur, 15 de Fevereiro de 1934) é um professor de informática suíço. Criador das linguagens de programação Pascal,2 Modula-2,3 e Oberon.

Graduado em engenharia eletrônica pelo Instituto Federal de Tecnologia de Zurique em 1959, M.Sc. na Universidade Laval em 1960, e Ph.D. na Universidade da Califórnia em Berkeley em 1963. Wirth foi um Professor/Assistente na ciência de computadores na Universidade de Stanford (1963 - 1967), e em seguida na Universidade de Zurique. Em 1968 tornou-se professor de informática na ETH Zurique. Ele permaneceu dois anos na Xerox PARC, na Califórnia, e aposentou-se em abril de 1999.

Resumo

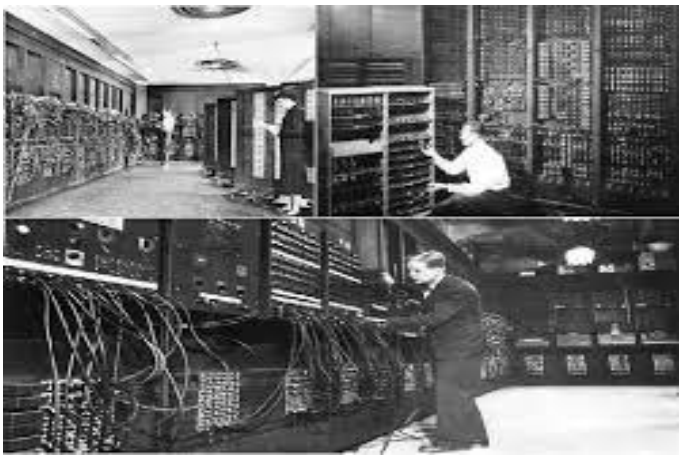
Nós apresentamos uma perspectiva pessoal da arte da programação. Começamos com o seu estado por volta de 1960 e acompanhamos o seu desenvolvimento até os dias atuais. O termo Engenharia de Software tornou-se conhecido após uma conferência em 1968, quando as dificuldades e armadilhas de projetar sistemas complexos foram discutidas francamente. A busca de soluções começou e se concentrou em melhores metodologias e ferramentas. As mais importantes foram as linguagens de programação que refletem os estilos procedimental, modular e, em seguida, orientado a objeto. A Engenharia de Software está intimamente ligada ao aparecimento e aperfeiçoamento desses estilos. Também importantes foram os esforços de sistematização, automatização da documentação do programa e testes. Por último, a verificação analítica e provas de correção deveriam substituir os testes.

Mais recentemente, o rápido crescimento do poder computacional tornou possível aplicar computação em tarefas cada vez mais complicadas. Esta tendência aumentou drasticamente as demandas por Engenharia de Software. Programas e sistemas se tornaram complexos e quase impossíveis de ser completamente compreendidos. A

queda dos custos e a abundância de recursos computacionais inevitavelmente reduziram os cuidados para um bom projeto. A qualidade parecia extravagante, uma perda na corrida pelo lucro. Mas devemos estar preocupados com a resultante deterioração da qualidade. Nossas limitações não são dadas por hardwares lentos, mas pela nossa própria capacidade intelectual. Por experiência, sabemos que a maioria dos programas pode ser significativamente melhorado, ficando mais confiáveis, econômicos e confortáveis de se utilizar.

A década de 1960 e a origem de Engenharia de Software

É lamentável que pessoas que lidam com os computadores costumam ter pouco interesse em sua história. Como resultado, muitos conceitos e idéias são propagados e anunciados como novos, sendo que existem há décadas, talvez sob uma terminologia diferente. Creio que vale a pena gastar ocasionalmente algum tempo para analisar o passado e investigar como os termos e conceitos surgiram.



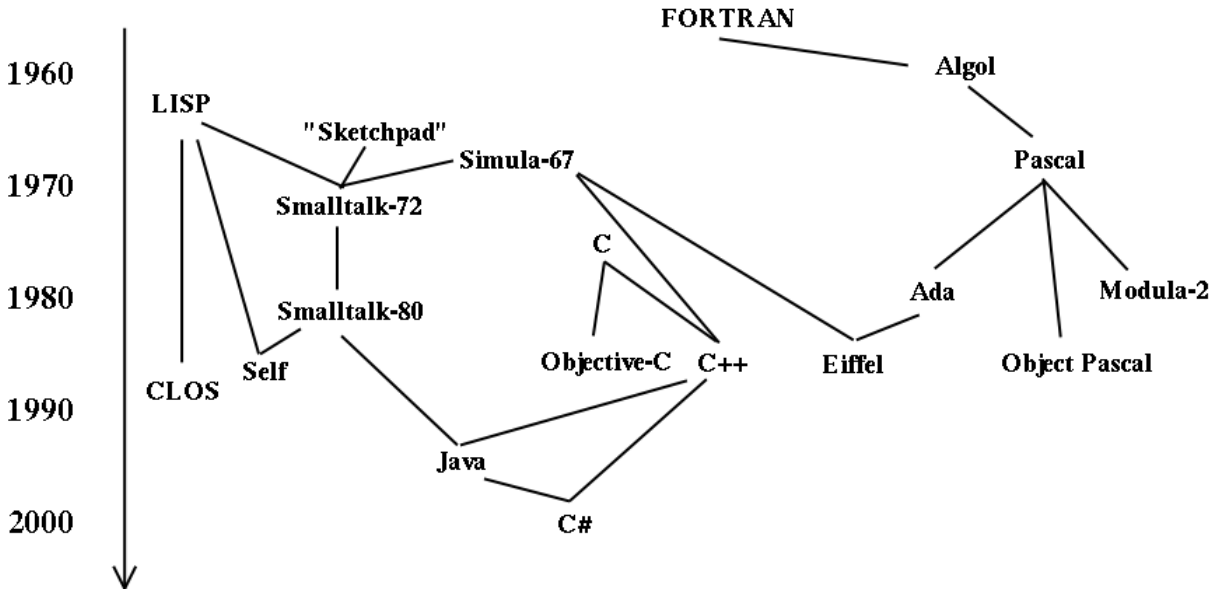
ENIAC – Primeiro computador eletrônico (década de

Eu considero o final dos anos 1950 como um período essencial da era da computação. Computadores de grande porte foram disponibilizados para instituições de pesquisa e

universidades. A presença deles foi notada principalmente na engenharia e nas ciências naturais, mas também nos negócios, onde logo tornaram-se indispensáveis. O tempo em que eles eram acessíveis apenas a uns poucos em laboratórios, quando quebravam toda vez que alguém queria usá-los, pertence ao passado. O seu aparecimento, dos laboratórios fechados de engenheiros eletricistas para o domínio público, fez com que a sua utilização, em especial a sua programação, se tornasse uma atividade para muitos. Uma nova profissão nasceu, mas os computadores de grande porte se tornaram ocultos, dentro de porões muito bem guardados. Programadores traziam os seus programas para o balcão, onde havia um atendente que os pegava e enfileirava, e onde os resultados poderiam ser buscados horas ou dias depois. Não havia nenhuma interatividade entre homem e computador.

UNIVAC I – Primeiro computador eletrônico de uso em negócios (década de 1950)

A programação foi conhecida por ser uma tarefa sofisticada que exige dedicação e pesquisas minuciosas, e uma paixão por códigos obscuros e truques. Para facilitar essa codificação, notações formais foram criadas. Nós agora as chamamos de linguagens de programação. A idéia principal era substituir seqüências



Histórico das linguagens de programação de alto nível

de código de instrução especial por fórmulas matemáticas. A primeira linguagem amplamente conhecida, Fortran, foi lançada pela IBM (Backus, 1957), logo seguida

pelo Algol (1958) e sua sucessora oficial em 1960. Como os computadores eram usados para a computação em vez de armazenamento de dados e comunicação, essas linguagens serviam principalmente para a cálculos matemáticos. Em 1962, a linguagem Cobol foi lançada pelo Departamento de Defesa dos EUA para aplicações de negócios.

Mas, como a capacidade de computação cresceu, assim também ocorreu com as exigências sobre os programas e programadores: tarefas tornaram-se mais e mais complicadas. Foi lentamente reconhecido que a programação era uma tarefa difícil, e que dominar os problemas complexos não era trivial, mesmo quando - ou porque - os computadores eram tão poderosos. A salvação foi procurada em "melhores" linguagens de programação, mais "ferramentas", sobretudo na automação. Uma melhor linguagem deve ser útil em uma ampla área de aplicação, ser mais parecida com uma linguagem "natural", oferecer mais facilidades. PL/1 foi concebida para unificar os mundos científico e comercial. Foi anunciada sob o slogan "Todo mundo pode programar graças ao PL/1". As linguagens de programação e seus compiladores tornaram-se um marco principal da Ciência da Computação. Mas eles não se ajustavam à matemática nem à eletrônica, os dois setores tradicionais em que os computadores eram usados. Uma nova disciplina surgiu, chamada de Ciência da Computação na América e de Informática na Europa.

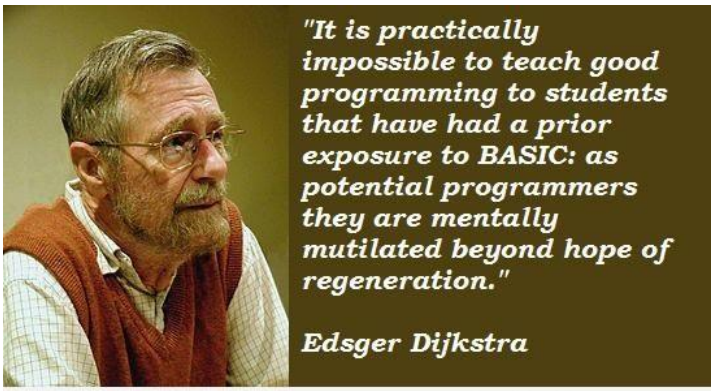
Em 1963, o primeiro sistema de tempo compartilhado apareceu (MIT, Stanford, McCarthy, DEC PDP-1). Isso trouxe de volta a interatividade. Os fabricantes de computadores aderiram à idéia e desenvolveram sistemas de tempo compartilhado para os seus grandes mainframes (IBM 360/67, a GE 645). Descobriu-se que a transição de sistemas de processamento em lote para sistemas de tempo compartilhado, ou a sua fusão, era muito mais difícil do que o previsto. Os sistemas eram anunciados e não podiam ser entregues a tempo. Os problemas eram muito complexos. As pesquisas deveriam ser conduzidasno dia-a-dia. As novidades do momento eram o multiprocessamento e programação concorrente. As dificuldades levaram as grandes empresas à beira do colapso. Em 1968, uma conferência patrocinada pela NATO, foi

IBM 360 (década de 1960)

dedicada ao tema (1968 em Garmisch-Partenkirchen, Alemanha) [1]. Apesar de críticas terem ocasionalmente sido expressas anteriormente [2, 3] só após a conferência as dificuldades foram discutidas abertamente e confessadas com franqueza incomum, e os termos "engenharia de software" e "crise de software" foram criados.

A programação como uma disciplina

No mundo acadêmico, foram sobretudo E.W. Dijkstra e C.A.R. Hoare que reconheceram os problemas e ofereceram novas idéias. Em 1965, Dijkstra escreveu o seu famoso "Notes on Structured Programming" [4] e declarou a programação como uma disciplina, em contraste com o artesanato. Também em 1965, Hoare publicou um importante artigo sobre estruturação de dados [5]. Essas idéias tiveram uma profunda influência sobre as novas linguagens de programação, em particular Pascal [6]. As línguagens são os veículos nos quais essas idéias deveriam ser expressas. A programação estruturada tornou-se sustentada por uma linguagem de programação estruturada.



Além disso, em 1966, Dijkstra escreveu um artigo seminal sobre processos que cooperam harmoniosamente [7], postulando uma disciplina baseada em semáforos como primitivas para sincronização de processos concorrentes. Hoare seguiu em 1966 com seu "Communicating Sequential Processes (CSP)" [8], percebendo que, no futuro, os programadores teriam que lidar com as dificuldades dos processos concorrentes. Obviamente, isso resultaria em uma metodologia estruturada e disciplinada ainda mais atraente.

Claro, tudo isso não mudou a situação, nem dissipou todas as dificuldades da noite pro dia. A indústria não poderia mudar nem as políticas nem as ferramentas rapidamente. No entanto, cursos de formação intensiva sobre a programação estruturada foram organizados, notavelmente através de H.D. Mills na IBM. Nada menos do que o Departamento de Defesa dos EUA percebeu que os problemas eram

urgentes e crescentes. Ele começou um projeto que culminou com a linguagem de programação Ada, uma linguagem altamente estruturada, apropriada para uma ampla variedade de aplicações. O desenvolvimento de software no Departamento de Defesa dos EUA seria então baseado exclusivamente em Ada [9].

Unix e C

No entanto, uma outra tendência começou a permear toda a área de programação, principalmente na academia, apontando na direção oposta. Foi provocada pela disseminação do sistema operacional UNIX, contrastando com o MULTICS do MIT e utilizado nos minicomputadores que estavam surgindo rapidamente. UNIX foi um alívio muito bem-vindo em relação aos grandes sistemas operacionais estabelecidos em computadores de grande porte. Em seu reboque, UNIX trouxe a linguagem C [8], que tinha sido expressamente concebida para apoiar o desenvolvimento do UNIX. Evidentemente por causa disso, era atrativo, senão mesmo obrigatório, o uso de C para o desenvolvimento de aplicações rodando em UNIX, que atuou como um cavalo de Tróia para C.

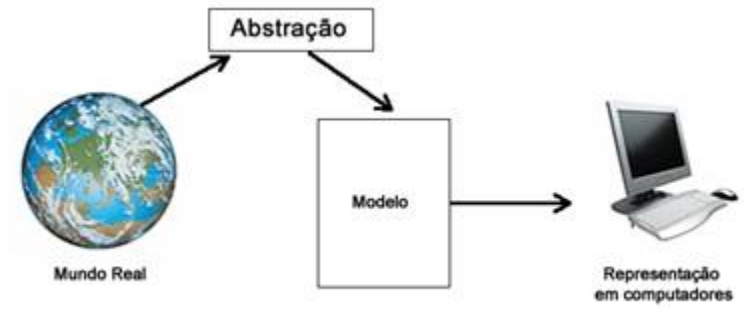
Do ponto de vista da Engenharia de Software, a rápida disseminação da C representou um grande salto para trás. Ele revelou que a comunidade em geral não havia compreendido o verdadeiro significado do termo "linguagem de alto nível", que se tornou um chavão mal-entendido. O que, então, deveria ser "alto nível"? Como esta questão está no centro de Engenharia de Software, precisamos entrar em detalhes.

Abstração

Os sistemas computacionais são máquinas de grande complexidade. Esta complexidade pode ser dominada intelectualmente por uma única ferramenta: Abstração.

Uma linguagem representa um computador abstrato cujos objetos e construções se encontram mais perto (em mais alto nível) para o problema a ser representado do que em uma máquina concreta. Por exemplo, em uma linguagem de alto nível lidamos com os números, matrizes indexadas, tipos de dados, instruções condicionais e repetitivas, e não com bits e bytes, palavras endereçadas, desvios e códigos de condição. No entanto, essas abstrações são benéficas somente se forem

consistentemente e completamente definidas em termos de suas próprias propriedades. Se isto não é assim, se as abstrações podem ser entendidas só em termos de facilidades de um computador subjacente, então os benefícios são marginais, quase insignifcantes. Se a depuração de um programa - sem dúvida a atividade mais comum da engenharia de software - requer uma "descarga da memória em hexadecimal", a linguagem vale pouco a pena.



A expansão generalizada do C mina a tentativa de elevar o nível da engenharia de software, porque C oferece abstrações que não se sustentam de fato: Matrizes permanecem sem a verificação de índice, os tipos de dados não são conferidos quanto a consistência, os ponteiros são meramente endereços onde adição e subtração são aplicáveis . Poderíamos ter classificado C como sendo algo entre enganoso e até mesmo perigoso. Mas, ao contrário, as pessoas em geral, particularmente na academia, acharam-no intrigante e "melhor que o código Assembly", porque ele apresentava alguma sintaxe.

O problema era que suas regras poderiam ser facilmente quebradas, exatamente o que muitos programadores estimavam. Era possível acessar todas as idiossincrasias de um computador, itens que uma linguagem de alto nível deveria esconder. C proporcionou liberdade, onde as linguagens de alto nível eram consideradas engessadas impondo uma disciplina indesejada. Era um convite para usar truques que tinham sido necessários para atingir a eficiência nos primeiros tempos dos computadores, mas agora eram armadilhas que tornavam grandes sistemas propensos a erros e dispendiosos para depurar e manter.

Linguagens que apareceram por volta de 1985 (como Ada e C + +), tentaram remediar estes defeitos e cobrir uma variedade muito maior de aplicações previsíveis. Como conseqüência, elas se tornaram grandes e suas descrições volumosas. Compiladores e ferramentas de apoio tornaram-se volumosos e complexos. Descobriu-se que, em vez de resolver problemas, eles acrescentaram problemas. Como Dijkstra disse: Eles pertenciam ao conjunto de problemas ao invés do conjunto de soluções.

Avanços na Engenharia de Software pareciam estagnar. As dificuldades cresceram mais rapidamente do que novos instrumentos que poderiam contê-las. No entanto, ao menos, pseudo-ferramentas como métricas de software revelaram-se como sendo de nenhuma ajuda, e os profissionais de Engenharia de Software já não eram julgados pelo número de linhas de código produzidas por hora.

O advento do micro computador

A propagação da Engenharia de Software e Pascal notadamente não ocorreu na indústria, mas em outras frentes: nas escolas e nas casas. Em 1975, micro-computadores apareceram no mercado (Commodore, Tandy, da Apple, muito mais tarde IBM). Eles foram baseados em processadores singlechip (Intel 8080, Motorola6800, a Rockwell 6502) com barramentos de 8 bits de dados, 32KB de memória ou menos, e frequências de relógio inferior a 1 MHz. Eles fizeram os computadores acessíveis às pessoas em contraste com as grandes organizações como empresas e universidades. Mas eram brinquedos e não máquinas computacionais úteis. O salto veio quando foi demonstrado que as linguagens poderiam ser usadas também com microcomputadores. O grupo de Ken Bowles da Universidade da Califórnia em San Diego construiu um editor de texto, um sistema de arquivos e um depurador de todo o compilador Pascal portátil (código P), desenvolvido na ETH, e os distribuiram por US$ 50. O mesmo fez a empresa Borland com a sua versão do compilador. Isso aconteceu numa época em que outros compiladores eram softwares caros, e foi nada menos do que uma virada na comercialização de software. De repente, havia um mercado de massa. A computação veio a público.

Microcomputador Commodore (década de 1970)

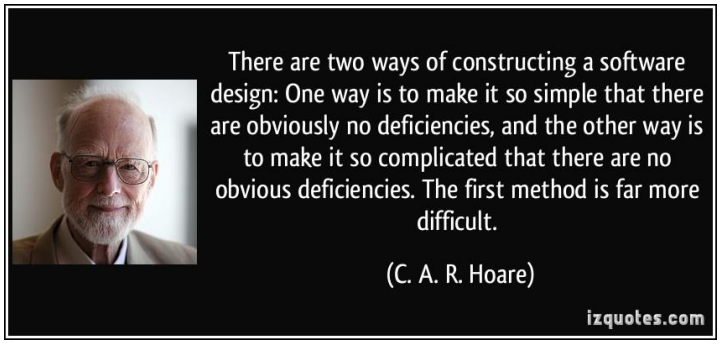

Microcomputador IBM PC (década de 1980)

Enquanto isso, as exigências em sistemas de software cresciam ainda mais, assim como a complexidade dos programas. O ofício da programação se tornou tarefa para invasores (“hackers”). Foram procurados métodos para sistematizar, se não a construção, pelo menos programas de testes e documentação. Embora isso fosse útil, os problemas reais de programação “agitada” sob pressão do tempo permaneceram. Dijkstra trouxe a dificuldade ao ponto de dizer: “Testes podem mostrar a presença de erros, mas nunca poderão provar a sua ausência”. Ele também desdenhou: “Engenharia de Software é a programação para quem não pode”.

A programação como uma disciplina matemática

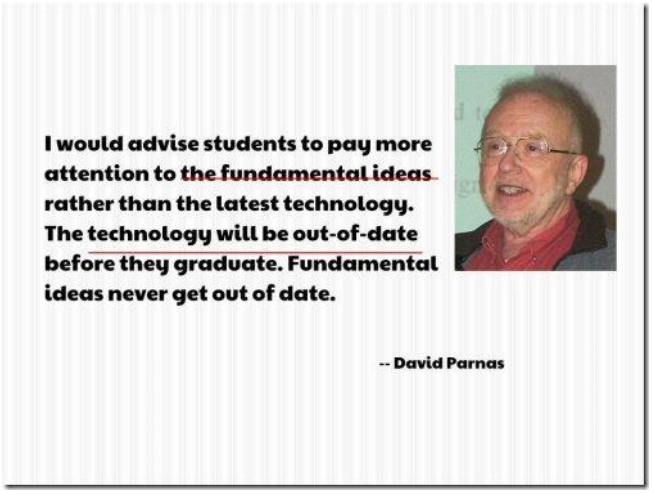
Já em 1968 R. W. Floyd sugeriu a ideia de asserções de estados, das verdades sempre válidas em certos pontos do programa [10]. Isso levou ao artigo seminal de Hoare intitulado "Uma base axiomática da Programação de Computadores", postulando a chamada lógica de Hoare [11]. Alguns anos mais tarde Dijkstra deduziu a partir dela, o cálculo de transformadores dos predicados [12]. A programação foi obtendo uma base matemática. Programas já não eram apenas o código para controle de computadores, mas textos estáticos, que podem ser submetidos a um raciocínio matemático.

Embora estes desenvolvimentos fossem reconhecidos em algumas universidades, eles passaram praticamente despercebidos na indústria. Na verdade, a lógica de Hoare e os transformadores de predicados de Dijkstra eram explicados de maneira satisfatória, mas para algoritmos simples, como a multiplicação de números inteiros, busca binária, e máximo divisor comum. Entretanto, a indústria foi atingida por sistemas verdadeiramente grandes. Isto não era óbvio para todos, se as teorias matemáticas iriam resolver problemas reais, quando a análise de algoritmos simples, por si só, já era bastante exigente.



A solução foi basear-se em uma forma disciplinada de programação, ao invés de em uma rigorosa teoria científica. Uma contribuição importante para a programação

estruturada foi feita por Parnas, em 1972, com a idéia de Ocultação de Informação [13], e ao mesmo tempo por Liskov com o conceito de Tipos de Dados Abstratos[14]. Ambos incorporam a idéia de quebrar sistemas de grande porte em partes, chamados módulos, e definir claramente as suas interfaces. Se um módulo A usa (importa) um módulo B, então A é chamado de um cliente de B. O projetista de A, então não precisa saber os detalhes, o funcionamento do B, mas apenas as propriedades declaradas em sua interface. Este princípio constitui, provavelmente, a mais importante contribuição à engenharia de software, ou seja, a construção de sistemas por grandes grupos de pessoas. O conceito de modularização é bastante reforçado pela técnica de compilação separada com verificação automática de compatibilidade nas interfaces.



Assim como a programação estruturada foi o espírito orientador do Pascal, a modularização foi a principal ideia por detrás da linguagem Modula-2, o sucessor do Pascal, publicado em 1979 [15]. De fato, sua motivação veio da linguagem Mesa, um desenvolvimento interno do Laboratório de Pesquisa da Xerox em Palo Alto, sendo ela mesma uma descendente do Pascal. O conceito de modularização e compilação em separado também foi adotado pela linguagem Ada (1984), que também foi amplamente baseada em Pascal. Nela os módulos foram chamados de pacotes.

A Era da estação de trabalho pessoal

No entanto, um outro desenvolvimento influenciou o campo de computação mais profundamente do que todas as linguagens de programação. Foi a estação de trabalho, cuja primeira encarnação, o “Alto”, foi construído, mais uma vez, no Laboratório de Pesquisa da Xerox em Palo Alto (1975) [16]. Em contraste com os referidos micro-computadores, a estação era poderosa o suficiente para permitir o desenvolvimento de software sério, computações complexas, bem como a utilização de um compilador para uma linguagem de programação avançada. O mais importante, foi pioneira nas telas de alta resolução – mapeamento de bits e no

dispositivo apontador chamado mouse, que, juntos, trouxeram uma mudança revolucionária no uso do computador. Junto com o “Alto”, o conceito de rede de área local (LAN) foi introduzido, bem como servidores centrais para impressão (a laser), armazenamento de arquivos em larga escala, e serviço de correio eletrônico. Não há exagero na afirmação que a era da computação moderna começou em 1975 com o “Alto”. O “Alto” causou nada mais nada menos do que uma revolução, e como resultado as pessoas de hoje não tem idéia de como a computação era feita antes de 1975 sem estações de trabalho pessoais altamente interativas. A influência desses acontecimentos sobre Engenharia de Software não pode ser sobrestimada.

Alto Xerox (década de 1970)

Como a demanda de softwares cada vez mais complexos cresceu persistentemente, como as dificuldades tornaram-se mais ameaçadoras e como alguns fracassos espetaculares demonstraram que os problemas eram graves, a procura de panaceias (remédio para todos os males) começou. Muitas curas foram oferecidas, vendidas, e logo esquecidas. Uma delas, no entanto, mostrou-se fecunda e sobreviveu: Programação orientada a objeto (POO).

Até 1980, o modelo de computação comumente aceito era transformar os dados de seu estado dado em resultado, transformando gradualmente a entrada em saída. Em sua forma abstrata mais simples, esta é a máquina de estado finito. Este ponto de vista da computação, surgiu a partir da tarefa original dos computadores: computação de resultados numéricos. No entanto, outro modelo ganhou terreno na década de 1960: era proveniente da simulação de sistemas complexos (supermercados, fábricas, ferrovias, logística). Sua abstração consiste de atores (processos) que vêm e vão, que passam fases em sua vida, e que trazem um conjunto de dados privados representando o seu estado atual. Provou-se natural pensar sobre tais atores com seus estados como uma unidade, como um objeto. Algumas linguagens de programação foram projetadas com base nesse modelo, sendo seu ancestral Simula, de Dahl e Nygaard em 1965. Mas elas permaneceram confinadas no campo de simulação de sistemas de eventos discretos. Somente após o surgimento de poderosos computadores pessoais que o modelo POO ganhou aceitação mais ampla. Agora, sistemas de computação possuem janelas, ícones, menus, botões, barras, etc, tudo facilmente identificável como objetos visíveis e com

estado e comportamentos individuais. Linguagens apareceram apoiando esse modelo, entre elas Smalltalk (Goldberg e Kay, 1980), Object Pascal (Tesler, 1985), C++ (Stroustrup, 1985), Oberon (Wirth, 1988), Java (Sun, 1995) e C # (Microsoft, 2000). A orientação a objeto tornou-se uma tendência e um chavão. De fato, escolher o modelo certo para uma aplicação é importante. Mesmo assim, não se deve desprezar o fato de que existem aplicações para as quais a POO não é o modelo adequado.

Abundância de Poder Computacional

O período desde 1985 até há alguns anos tem se caracterizado principalmente por enormes avanços na tecnologia de hardware. Hoje, mesmo minúsculos computadores, tais como os telefones celulares, têm potência e capacidade cem vezes maiores do que tinham 20 anos atrás. É justo dizer que as tecnologias de semicondutores e de discos têm determinado todos os avanços recentemente. Quem, por exemplo, teria sonhado em 1990 com pastilhas de memória com vários gigabytes de dados, discos minúsculos com dezenas de gigabytes de capacidade e processadores com taxas de relógio de vários gigahertz?


Vint Cerf e Robert Kahn criadores do protocolo TCP/IP, o componente de software fundamental da Internet

Este rápido desenvolvimento ampliou significativamente a área de aplicações do computador. Isto aconteceu sobretudo em relação a tecnologia de comunicação. Agora, é difícil acreditar que antes de 1975 tecnologias de comunicação e computação foram consideradas campos distintos. A eletrônica as uniu e a Internet cresceu. Sua característica é uma largura de banda que parece ser ilimitada. Fico impressionado quando comparo isso com o primeiro minicomputador

Timothy Berners-Lee criador da World Wide Web - www

com que eu trabalhei em 1965, um DEC PDP-1: taxa de relógio <1 MHz, memória de 8K palavras de 18 bits e um tambor de armazenamento de cerca de 200 KB. Ele era compartilhado por até 16 usuários. É um milagre que algumas pessoas insistiam em acreditar que um dia os computadores se tornariam poderosos o suficiente para serem úteis.

Na década de 1990, um fenômeno começou a se espalhar sob o nome de Código Aberto. A desconfiança contra os grandes sistemas projetados em segredo industrial se manifestou. A vasta comunidade de programadores decidiu construir softwares e distribuir seus produtos gratuitamente através da Internet. Embora seja difícil reconhecer isso como um princípio de negócio - tornando a idéia de patentes obsoleta - o movimento acabou por ser bastante bem sucedido. As noções de qualidade e responsabilidade em caso de falha pareciam irrelevantes. O Código Aberto apareceu como a alternativa bem-vinda à hegemonia industrial e ao lucro abrasivo, e também contra a dependência desamparada.

Precursores do movimento Open Source

É geralmente difícil em Engenharia de Software distinguir as estratégias de negócios das idéias científicas.Nessas últimas, o Código Aberto parece ser uma última tentativa de encobrir o fracasso. A escrita de código complicado e a desagradável descriptografia por outros é aparentemente considerada mais fácil ou mais econômica do que o projeto cuidadoso e a descrição de interfaces claras dos módulos. A fácil adaptação dos módulos, quando disponíveis em código fonte é também um argumento fraco. Qual seria o interesse em um crescimento selvagem das variedades das variantes? Não o da engenharia de alta qualidade e do profissionalismo.

Desperdício de Software

Enquanto o incrível aumento no poder de hardware foi muito benéfico para uma ampla gama de aplicações (pensamos em administração, bancos, ferrovias, companhias aéreas, sistemas de orientação, engenharia, ciência), o mesmo não pode ser dito em relação à engenharia de software. Certamente, a Engenharia de Software

tem se beneficiado também das muitas ferramentas sofisticadas de desenvolvimento. Mas a qualidade de seus produtos dificilmente reflete sinais de grande progresso. Não é à toa: afinal, o aumento do próprio poder foi a razão para o crescimento assustador da complexidade. Qualquer que seja o progresso feito na metodologia de software, este será rapidamente compensado pela maior complexidade das tarefas. Isto é refletido pela Lei de Reiser: "O software se torna lento mais rapidamente do que o hardware se torna rápido". Na verdade, novos problemas têm sido resolvidos, mas são tão difíceis que os engenheiros têm muitas vezes de ser admirados mais por seu otimismo e coragem do que por seu sucesso.

O que aconteceu em Engenharia de Software era previsível e inerente a um campo da engenharia, onde a demanda cresce, o trabalho é feito sob pressão (de tempo), e o custo dos recursos está quase desaparecendo. A conseqüência é o desperdício de recursos baratos - ciclos de processador e bits de armazenamento - resultando em um código ineficiente e dados volumosos. Este desperdício se torna cada vez mais presente e representa uma grave falta de senso de qualidade. A ineficiência dos programas é facilmente coberta pela obtenção de processadores mais rápidos, e o projeto precário da estrutura de dados é compensado pela utilização de dispositivos de armazenamento maiores. Mas seus efeitos colaterais são a diminuição da qualidade - de robustez, confiabilidade e facilidade de uso. Bom, um projeto cuidadoso é demorado e dispendioso. Mas ainda é mais barato do que o do não confiável e complicado, quando o custo de "manutenção" não está contabilizado A tendência é inquietante, e assim é a complacência de clientes.

Reflexões Pessoais e Conclusões

O que podemos fazer para liberar essa sobrecarga? Há pouca vantagem em ler a história, a menos que estejamos dispostos a aprender com ela. Por isso, atrevo-me a refletir sobre o passado e tentar tirar algumas conclusões. Um esforço principal deve ser a educação com um sentido de qualidade.

Os desenvolvedores de software devem estar engajados na cruzada contra a complexidade de fabricação caseira. O crescimento canceroso da complexidade não é uma coisa para ser admirada, ele deve ser combatido sempre que possível [17]. Programadores devem dispor de tempo e respeito para produzir um trabalho de alta qualidade. Isso é fundamental e, ultimamente, é mais eficaz do que as melhores

ferramentas e regras. Vamos iniciar um esforço global para impedir que o software se torne conhecido como *softwaste*!

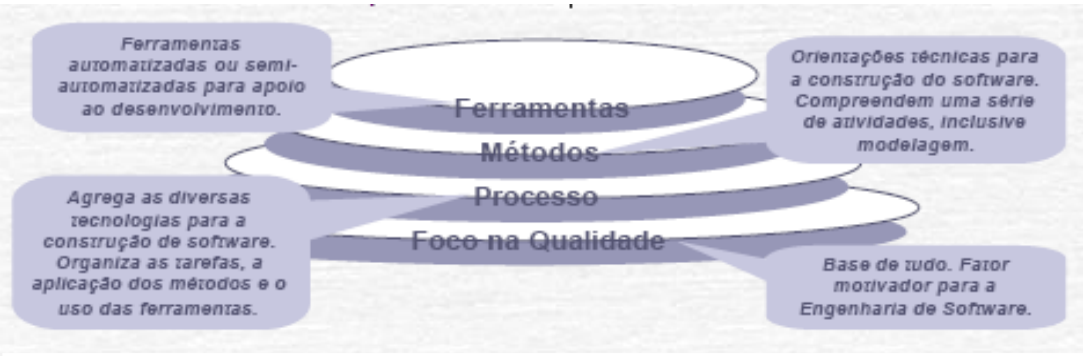
Recentemente eu tenho me familiarizado com alguns projetos onde grandes sistemas operacionais comerciais foram descartados em favor do Sistema Oberon, cujo principal objetivo foi a clareza e a concentração no essencial [18]. Os líderes do projeto, sendo obrigados a entregar um software confiável e econômico, reconheceram que eram incapazes de fazê-lo, - mesmo com todo o cuidado – tendo que construir o seu trabalho em cima do software de base complexo - uma plataforma - que nem era totalmente descrita, nem segura. Nós sabemos que qualquer corrente é tão forte quanto seu elo mais fraco. Isso vale também para as hierarquias de módulos. Os sistemas podem ser projetados com cuidado e profissionalismo, mas continuarão sujeitos a erros se construídos sobre uma plataforma complexa e pouco confiável.

A louca corrida por uma maior complexidade - eufemisticamente chamada de sofisticação - há muito tempo também se apropriava do instrumento mais importante do engenheiro de software. Linguagens modernas como Java e C # podem ser melhores do que as antigas como Fortran, PL / I e C, mas estão longe de serem perfeitas, e elas poderiam ser muito melhores. Seus manuais de várias centenas de páginas são um sintoma inequívoco da sua inadequação. Engenheiros na indústria, no entanto, raramente são livres de restrições. Supostamente, eles devem ser compatíveis com o resto do mundo, e se desviar dos padrões estabelecidos pode ser fatal.

Mas isso não pode ser dito sobre as universidades. É, portanto, um fato triste que elas têm permancido inativas e complacentes. Não só tem a pesquisa em linguagem e metodologia de projeto perdido o seu glamour e atratividade, mas pior, as ferramentas comuns na indústria tem sido discretamente adotadas sem debate e crítica. As linguagens atuais podem ser inevitáveis na indústria, mas para ensinar, para uma introdução fundamentada, ordenada, estrutrada e sistemática, elas são totalmente erradas e obsoletas.

Isto está notavelmente de acordo com as tendências do século 21: Nós ensinamos, aprendemos e realizamos apenas o que é imediatamente rentável, o que é solicitado pelos alunos. Em poucas palavras: Nós focamos no que vende. Universidades eram tradicionalmente isentas desta corrida comercial. Eram lugares onde as pessoas deviam refletir sobre o que interessa a longo prazo. Elas eram líderes

espirituais e intelectuais, mostrando o caminho para o futuro. Em nossa área de computação, estou com medo, elas simplesmente tornam-se seguidores dóceis. Elas



O foco da Engenharia de Software deve ser a Qualidade de seus processos e produtos

parecem ter sucumbido ao anseio da moda para a inovação contínua, e ter perdido de vista a necessidade do trabalho cuidadoso.

Se podemos aprender alguma coisa com o passado, é que a ciência da computação é na essência uma questão metodológica. É suposto desenvolver técnicas (ensináveis) e conhecimento que são geralmente benéficos em uma ampla variedade de aplicações. Isso não significa que a Ciência da Computação deva derivar para todas estas aplicações diversas e acabe perdendo a sua identidade. A Engenharia de Software seria a principal beneficiária de uma educação profissional em programação disciplinada. Entre suas feramentas, as linguagens figuram na vanguarda. Uma linguagem com construções adequadas e estrutura, suportada por abstrações limpas, contribui para a construção de artefatos e é essencial na educação. A complexidade caseira e artificial não tem lugar entre essas linguagens. E finalmente: Deve ser um prazer trabalhar com elas, porque elas nos permitem criar artefatos que podemos mostrar e nos orgulhar.

Referencias

1. P. Naur and B. Randell, Eds. *Software Engineering*. Report on a Conference held in Garmisch, Oct. 1968, sponsored by NATO

2. E.W. Dijkstra. Some critical comments on advanced programming. *Proc. IFIP Congress,* Munich, Aug. 1962.

3. R.S. Barton. A critical review of the state of the programming art. *Proc. Spring Joint Computer Conference*, 1963, pp 169 – 177.

4.. E. W. Dijkstra. Notes on structured programming. In *Structured Programming*. O.-J. Dahl, E. W. Dijkstra and C.A.R. Hoare, Acad. Press, 1972.

5. C.A.R. Hoare. Notes on data structuring. In *Structured Programming*. O.-J. Dahl, E. W. Dijkstra and C.A.R. Hoare, Acad. Press, 1972.

6. N. Wirth. The Programming Language Pascal. *Acta Informatica 1*, (1971) 35 - 63

7. E. W. Dijkstra, Cooperating sequential processes. Sept. 1965. Reprinted in *Programming Languages*, F. Genuys, Ed., Acad. Press, New York, 1968, 43-112.

8. C.A.R. Hoare. Communicating sequential processes *Comm. ACM, 21*, 8 (August 1978) pp. 666 - 677.

9. J.G.P. Barnes. An Overview of Ada. *Software - Practice and Experience, 10* (1980) 851 – 887.

10. R.W. Floyd. Assigning meanings to programs*, Proc. of Symp. in Applied Mathematics., 19* (1967), pp. 19-32

11. C.A.R. Hoare**.** An axiomatic basis for computer. *Comm. ACM, 12*, 10 (October 1969), pp. 576 - 580

12. E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Comm. ACM, 18*, 8, pp. 453–457, August 1975.

13. D. L. Parnas. Abstract types defined as classes of variables. *ACM Sigplan Notices II 2*, 149 - 154 (1976)

14. B. Liskov and S. Zilles**.** Programming with abstract data types**.** *Proc. ACM SIGPLAN symposium*, Santa Monica, 1974, pp. 50-59.

15. N. Wirth. *Programming in Modula-2.* Springer, 1974. ISBN 0-387-50150-9.

16. C.P. Thacker et al. Alto: A personal computer. Xerox PARC, Tech. Rep CSL-79-11

17. N. Wirth. A plea for lean software. IEEE *Computer*, Feb. 1995, pp. 64-68.

18. M. Franz. Oberon: The overlooked Jewel. In L. Boszormenyi, J. Gutknecht, G. Pomberger. *The School of Niklaus Wirth*. ISBN 1-55860-723-4 and 3-932588-85-1.