As Leis Fundamentais do Projeto de Software

Max Kanat-Alexander

Novatec

Editor: Rubens Prates
Tradução: Luís Euclides dos Santos
Revisão técnica: Joel Saade
Revisão gramatical: Giacomo Leone Neto
Editoração eletrônica: Carolina Kuwabata

ISBN: 978-85-7522-302-4

Histórico de impressões:

Setembro/2012 Primeira edição

Novatec Editora Ltda.
Rua Luís Antônio dos Santos 110
02460-000 – São Paulo, SP – Brasil
Tel.: +55 11 2959-6529
Fax: +55 11 2950-8869
E-mail: novatec@novatec.com.br
Site: www.novatec.com.br
Twitter: twitter.com/novateceditora
Facebook: facebook.com/novatec
LinkedIn: linkedin.com/in/novatec

Dados Internacionais de Catalogação na Publicação (CIP)
(Câmara Brasileira do Livro, SP, Brasil)

Kanat-Alexander, Max
As leis fundamentais do projeto de software / Max Kanat-Alexander ; tradução Luís Euclides dos Santos. -- São Paulo : Novatec Editora ; Sebastopol, CA : O'Reilly,, 2012.

Título original: Code simplicity.
ISBN 978-85-7522-302-4

1. Engenharia de software 2. Programa de computação 3. Software - Desenvolvimento 4. Teoria da codificação I. Título.

12-11080 CDD-005.13

Índices para catálogo sistemático:

1. Desenvolvimento de software : Computadores : Processamento de dados 005.13

MKP20120829

CAPÍTULO 1

Introdução

Os computadores causaram uma importante mudança social. A razão é que eles nos permitem fazer mais trabalho com menos pessoas. Esse é o valor de um computador – ele pode fazer muito trabalho, realmente rápido.

Isso é ótimo.

O problema é que os computadores quebram. Eles quebram o tempo todo. Se qualquer outra coisa em sua casa quebrasse com tanta frequência quanto seu computador, você a devolveria. A maioria das pessoas nas sociedades modernas passa pela experiência de ter um computador quebrado ou se comportando mal para elas pelo menos uma vez ao dia.

Isso não é tão ótimo.

O que há de errado com os computadores?

Por que os computadores quebram tanto? Para o software, há uma razão, e somente uma razão: má programação. Algumas pessoas culpam a administração e outras culpam os clientes, mas a investigação mostra que a raiz do problema é sempre a programação.

Mas o que queremos dizer com "má programação"? Esse é um termo muito ambíguo. E os programadores geralmente são pessoas muito inteligentes e racionais – por que alguns deles fariam "má" programação?

Basicamente, tudo gira em torno da complexidade.

O computador é provavelmente o dispositivo mais complexo que podemos produzir em uma fábrica hoje. Ele faz bilhões de cálculos por segundo. Ele tem centenas de milhões de minúsculas partes eletrônicas que devem operar corretamente a fim de que ele funcione.

Um programa escrito em um computador é igualmente complexo. Por exemplo, quando foi escrito, o Microsoft Windows 2000 era um dos maiores programas já criados, tendo em torno de 30 milhões de linhas de código. Escrever tanto código assim é algo como escrever um livro de 200 milhões de palavras – mais de cinco vezes o tamanho da Enciclopédia Britânica.

A complexidade de um programa pode particularmente confundir, porque não há nada em que colocar suas mãos. Quando ele falha, você não pode pegar algo sólido e examinar por dentro. É tudo abstrato, e isso pode ser realmente difícil de se lidar. Na verdade, o programa de computador médio é tão complexo que nenhuma pessoa poderia compreender como todo o código funciona em sua totalidade. Quanto maiores ficam os programas, mais esse é o caso.

Assim, programar tem que se tornar o ato de reduzir complexidade à simplicidade. Caso contrário, ninguém poderia continuar trabalhando em um programa depois que ele alcançasse certo nível de complexidade. As partes complexas de um programa têm que ser organizadas de algum modo simples para que um programador possa trabalhar nele sem ter capacidades mentais divinas.

Essa é a arte e o talento envolvidos na programação – reduzir complexidade à simplicidade.

Um "mau programador" é apenas alguém que falha em reduzir a complexidade. Muitas vezes isso acontece porque as pessoas acreditam que estão reduzindo a complexidade de escrever na linguagem de programação (o que é definitivamente por si só toda uma complexidade) ao escrever código que "apenas funciona", sem pensar em reduzir a complexidade para outros programadores.

É mais ou menos assim.

Imagine um engenheiro que, precisando de algo com que pregar um prego no chão, inventa um dispositivo envolvendo polias, cordas e um grande ímã. Você provavelmente pensaria que isso seria bastante ridículo.

Agora, imagine que alguém lhe diz: "Preciso de algum código que eu possa usar em qualquer programa, em qualquer lugar, que se comunicará entre quaisquer dois computadores, usando qualquer meio imaginável". Isso é definitivamente mais difícil de reduzir a algo simples. Então, alguns programadores (talvez a maior parte dos programadores) nessa situação surgirão com uma solução que envolve o equivalente a cordas, polias e um grande ímã, o que não seria muito compreensível para outras pessoas. Eles não são irracionais e não há nada errado com eles. Quando confrontados com uma tarefa realmente difícil, eles farão o que puderem no curto tempo que tiverem. O que eles fizerem, funcionará, no que depender deles. Fará o que se espera que faça. Isso é o que o chefe deles quer, e é o que os seus clientes parecem querer, também.

Mas, de um modo ou de outro, eles terão falhado em reduzir a complexidade à simplicidade. Então eles passarão esse dispositivo a outro programador e este contribuirá para a complexidade usando-o como parte de seu dispositivo. Quanto maior o número de pessoas que não agem para reduzir a complexidade, mais incompreensível o programa se torna.

À medida que um programa se aproxima da complexidade infinita, torna-se impossível encontrar todos os problemas com ele. Aviões a jato custam milhões ou bilhões de dólares porque estão próximos desse complexo e foram "depurados". Mas a maioria dos softwares custa ao cliente cerca de $50/$100. A esse preço, ninguém vai ter tempo ou recursos necessários para eliminar todos os problemas de um sistema infinitamente complexo.

Então, um "bom programador" deve fazer tudo que estiver ao seu alcance para tornar o que escreve o mais simples possível para outros programadores. Um bom programador cria coisas que são fáceis de entender, para que seja realmente fácil eliminar todos os erros.

Agora, às vezes, essa ideia de simplicidade é mal compreendida significando que os programas não devem ter muito código, ou que não devem usar tecnologias avançadas. Mas isso não é verdade. Às vezes, muito código realmente leva à simplicidade; significa apenas mais escrita e mais leitura, o que é bom. Você tem de se certificar de que exista algum documento breve que explique a grande quantidade de código, mas isso é tudo parte da redução da complexidade. Além disso, normalmente, tecnologias mais avançadas levam a mais simplicidade, mesmo que você tenha que aprender sobre elas primeiro, o que pode ser inconveniente.

Algumas pessoas acreditam que escrever de modo simples leva mais tempo do que escrever rapidamente algo que "faz o serviço". Na verdade, gastar um pouco mais de tempo escrevendo código simples revela-se mais rápido do que escrever muito código rapidamente no início e então gastar muito tempo tentando entendê-lo mais tarde. Essa é uma simplificação muito grande da questão, mas a história da indústria da programação tem mostrado que esse é o caso. Muitos programas ótimos tiveram seu desenvolvimento interrompido no decorrer dos anos só porque levou muito tempo para adicionar recursos às complexas bestas que eles haviam se tornado.

E é por isso que os computadores falham com tanta frequência – porque na maior parte dos programas existentes, muitos dos programadores da equipe falharam ao reduzir a complexidade das partes do que estavam escrevendo. Sim, é difícil. Mas não é nada em comparação com a infindável dificuldade que os usuários vivenciam quando têm que usar sistemas complexos e com falhas projetados por programadores que erraram em simplificar.

O que é um programa, realmente?

A frase "um programa de computador", do jeito que a maioria das pessoas a usa, carrega duas definições muito distintas:

1. Uma sequência de instruções dadas ao computador.
2. As ações executadas por um computador como resultado de ter recebido instruções.

A primeira definição é o que os programadores veem quando estão escrevendo um programa. A segunda definição é o que os usuários veem quando estão usando um programa. O programador diz ao computador: "Exiba um porco na tela". Essa é a definição 1, algumas instruções. O computador espalha muita eletricidade que faz com que um porco apareça na tela. Essa é a definição 2, as ações executadas pelo computador. Tanto o programador quanto o usuário aqui diriam que estão trabalhando com "um programa de computador", mas suas experiências com ele são muito diferentes. Programadores trabalham com palavras e símbolos, enquanto usuários veem apenas o resultado final – as ações executadas.

Em última análise, um programa de computador é essas duas coisas: as instruções que o programador escreve a as ações que o computador executa. O objetivo de escrever as instruções é fazer com que as ações aconteçam – sem as ações, não haveria razão para escrever as instruções. É exatamente como na vida, quando você escreve uma lista de compras. Esta é um conjunto de instruções sobre o que comprar na loja. Se você apenas escrevesse as instruções, mas nunca fosse até a loja, isso seria completamente sem sentido. As instruções precisam fazer com que algo aconteça.

Contudo, há uma diferença significativa entre escrever uma lista de compras de gêneros alimentícios e escrever um programa de computador. Se sua lista de gêneros alimentícios for desorganizada, isso apenas retardará ligeiramente sua experiência de compra. Mas, se o código de seu programa for desorganizado, atingir suas metas pode tornar-se um pesadelo. Por que isso? Bem, listas de mercearia são curtas e simples, e você as joga fora quando acaba de usá-las. Programas de computador são grandes e complexos, e você com frequência precisa mantê-los por anos ou décadas. Então, embora uma lista de mercearia curta e simples possa trazer pouca dificuldade a você, não importa o quanto ela esteja desorganizada, mas se um código de programa for desorganizado, poderá causar a você grandes dificuldades.

Adicionalmente, não há outra área em que um conjunto de instruções e o resultado dessas instruções estejam tão ligados quanto na área de desenvolvimento de software. Em outras áreas, as pessoas escrevem instruções e depois as entregam a outras, frequentemente esperando muito tempo

para vê-las executadas. Por exemplo, quando um arquiteto projeta uma casa, ele escreve um conjunto de instruções – os blueprints (desenhos). Estes devem passar por muitas pessoas diferentes no decorrer de muito tempo para tornar-se uma casa real. A construção final é o resultado das interpretações de todas essas pessoas das instruções do arquiteto. Por outro lado, quando escrevemos código, não há ninguém entre nós e o computador. O resultado é exatamente o que as instruções disseram para fazer, sem questionar. A qualidade do resultado final depende inteiramente da qualidade da máquina, de nossas ideias e de nosso código.

Desses três fatores, a qualidade do código é o maior problema enfrentado nos projetos de software hoje. Por causa desse fato e de todos os outros pontos mencionados, a maior parte deste livro é sobre como melhorar a qualidade do código. Abordamos ideias e máquinas também em alguns lugares, mas no geral o foco é em melhorar a estrutura e a qualidade das instruções que você está fornecendo à máquina.

Quando gastamos tanto tempo conversando sobre código, contudo, é muito fácil esquecer que estamos fazendo isso porque desejamos um resultado melhor. Nada neste livro perdoa um resultado ruim – toda a razão de focarmos em melhorar o código é porque melhorar o código é o problema mais importante que devemos resolver a fim de melhorar o resultado.

O que mais precisamos, então, é de uma ciência para melhorar a qualidade do código.