

As Leis Fundamentais do Projeto de Software

O bom projeto de software é simples e fácil de entender. Infelizmente, o programa de computador médio hoje é tão complexo que ninguém poderia possivelmente compreender como todo o código funciona. Este guia conciso ajuda você a entender os fundamentos do bom projeto por meio de leis científicas – princípios que você pode aplicar a qualquer linguagem de programação ou projeto daqui até a eternidade.

Se você é um programador júnior, engenheiro de software sênior ou gerente não técnico, aprenderá a criar um plano sólido para seu projeto de software e a tomar decisões melhores sobre o padrão e a estrutura de seu sistema.

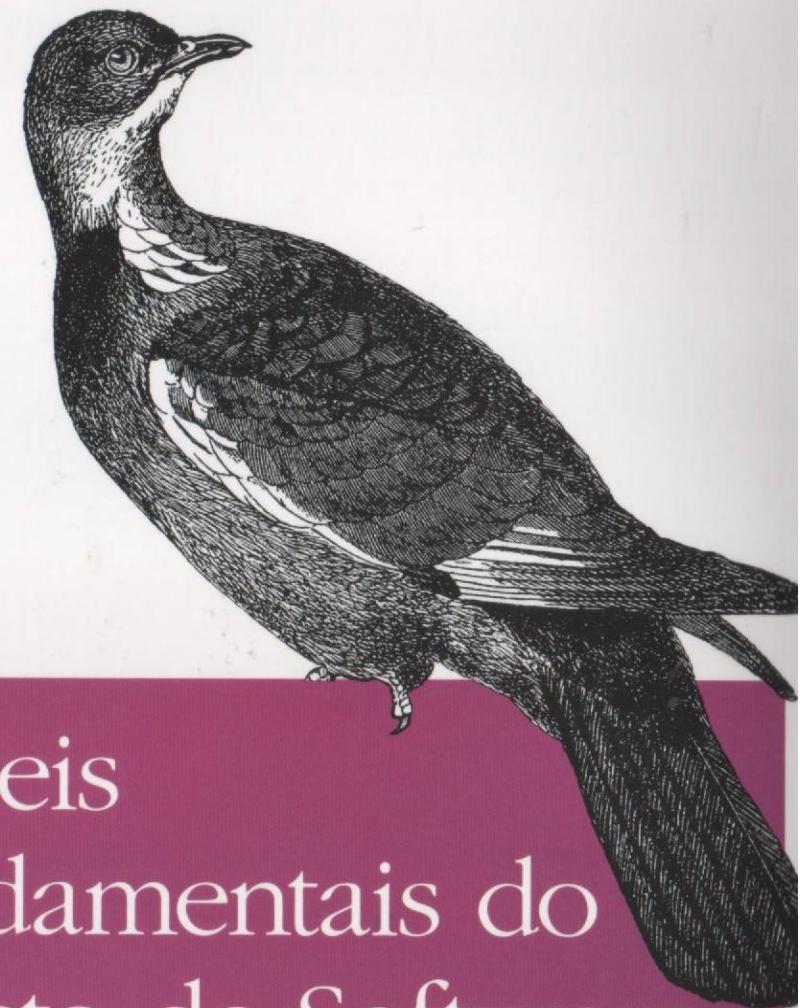
- Descubra por que o bom projeto de software se tornou a ciência ausente
- Entenda o propósito definitivo do software e os objetivos do bom projeto
- Determine o valor de seu projeto agora e no futuro
- Examine exemplos do mundo real que demonstram como um sistema muda com o tempo
- Crie projetos que permitam a máxima alteração no ambiente com a menor alteração no software
- Faça alterações mais fáceis no futuro mantendo seu código mais simples agora
- Obtenha melhor conhecimento sobre o comportamento de seu software com testes mais precisos

As Leis Fundamentais do Projeto de Software

novatec ■ O'REILLY

Fique conectado:

-  twitter.com/novateceditora
-  facebook.com/novatec
-  www.novatec.com.br



As Leis Fundamentais do Projeto de Software

O'REILLY®
novatec

Max Kanat-Alexander

Prefácio

A diferença entre um mau e um bom programador é o entendimento. Isto é, maus programadores não entendem o que estão fazendo, e bons programadores entendem. Acredite ou não, é realmente simples assim.

Este livro existe para ajudar todos os programadores a entender o desenvolvimento de software em um nível bem amplo que pode ser aplicado a qualquer linguagem de programação ou projeto daqui até a eternidade. Ele apresenta leis científicas para desenvolvimento de software, de uma forma simples que qualquer pessoa consegue ler.

Se você é programador, essas leis ajudarão a explicar por que certos métodos de desenvolvimento de software funcionam e por que alguns não. Elas ajudarão a guiá-lo na tomada de decisões sobre desenvolvimento de software com base no dia a dia, e ajudarão sua equipe a ter conversas inteligentes que levam a planos sensatos.

Se você não é programador, mas trabalha na indústria de software, poderá achar este livro útil por várias razões:

- É uma excelente ferramenta educacional para uso no treinamento de programadores iniciantes, contendo ainda informações altamente relevantes para programadores experientes.
- Ele permitirá que você entenda de modo mais eficaz por que os engenheiros de software querem fazer certas coisas, ou por que o software deve ser desenvolvido de certo modo.
- Ele pode ajudá-lo a transmitir suas ideias de modo eficaz a engenheiros de software, ajudando-o a entender os princípios fundamentais nos quais os bons engenheiros de software baseiam suas decisões.

Em tese, todos que trabalham na indústria de software devem ser capazes de ler e entender este livro, mesmo que não tenham muita experiência em programação, ou mesmo que o inglês não seja seu idioma nativo. Ter mais entendimento técnico ajudará a compreender alguns dos conceitos, mas a maior parte não exige nenhuma experiência em programação para ser entendida.

Na verdade, mesmo sendo este livro sobre desenvolvimento de software, ele não contém quase nenhum código de programa. Como pode ser isso? Bem, a ideia é que esses princípios devem se aplicar a qualquer projeto de software, em qualquer linguagem de programação. Você não precisa saber alguma linguagem de programação específica apenas para entender coisas que se aplicam a toda programação, em toda parte. Em vez disso, exemplos do mundo real e analogias são usados ao longo do livro para ajudá-lo a obter um melhor entendimento de cada princípio à medida que ele é apresentado.

Acima de tudo, este livro foi escrito para ajudá-lo e para ajudar a trazer bom-senso, ordem e simplicidade à área de desenvolvimento de software. Espero que você aprecie lê-lo e melhore sua vida e seu software de algum modo.

Convenções usadas neste livro

As seguintes convenções tipográficas são usadas neste livro:

Itálico

Indica termos novos, URLs, endereços de email, nomes e extensões de arquivos.

Largura constante

Usada em listagens de programas e também dentro de parágrafos para se referir a elementos de programas, tais como nomes de variáveis ou funções.



Esse ícone significa uma dica, sugestão ou observação geral.

Atribuição e licenças

Este livro está aqui para ajudá-lo a fazer seu trabalho. Se você fizer referências a partes limitadas em seu trabalho ou em seus escritos, apreciamos, mas não exigimos atribuição. Uma atribuição normalmente inclui o título, autor, editor e ISBN. Por exemplo: "Code Simplicity: The Science of Software Development by Max Kanat-Alexander (O'Reilly). Copyright 2012 Max Kanat-Alexander, 978-1-4493-1389-0".

Se você sente que seu uso de exemplos ou citações deste livro não corresponde a um uso justo ou à licença concedida, fique à vontade para entrar em contato conosco em permissions@oreilly.com.

Como entrar em contato conosco

O autor mantém um site em <http://www.codesimplicity.com>, por meio do qual você pode fazer contribuições e enviar comentários e correções.

Envie seus comentários e questões sobre este livro à editora escrevendo para: novatec@novatec.com.br.

Temos uma página web para este livro, na qual incluímos erratas, exemplos e quaisquer outras informações adicionais.

- Página da edição em português
http://www.novatec.com.br/catalogo/7522302_leisprosoft
- Página da edição original em inglês
<http://www.oreilly.com/catalog/9781449313890>

Para obter mais informações sobre os livros da Novatec, acesse nosso site em: <http://www.novatec.com.br>.

Agradecimentos

Meus editores, Andy Oram e Jolie Kanat, foram um recurso inestimável. O feedback de Andy foi tanto cheio de insights quanto brilhante. A insistência e o apoio de Jolie fizeram definitivamente com que este livro fosse publicado, e o seu extenso trabalho de revisão nos rascunhos iniciais foi bastante apreciado.

Minha copyeditor, Rachel Head, tem um talento notável para esclarecer e melhorar tudo.

Todos os programadores com os quais trabalhei e conversei na comunidade open source também merecem agradecimentos – particularmente meus colegas desenvolvedores no Projeto Bugzilla, que me ajudaram a experimentar todas as ideias deste livro em um sistema de software real e ao vivo no decorrer de muitos anos.

Os comentários e feedback que recebi em meu blog ao longo dos anos me ajudaram a moldar a forma e o conteúdo deste livro. Todas as pessoas que participaram dele merecem agradecimentos, mesmo aquelas que simplesmente me incentivaram ou me informaram que haviam lido um artigo.

Em um nível pessoal, sou muito grato a Jevon Milan, Cathy Weaver e a todos que trabalham com eles. Em um sentido bem verdadeiro, eles são responsáveis por eu ser capaz de escrever este livro. E, finalmente, tiro o chapéu para o meu amigo Ron, sem o qual este livro nem mesmo teria sido possível.

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 Encyclopédia 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 infinidável 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 e 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.

CAPÍTULO 2

A ciência ausente

Este livro é em sua maior parte sobre algo chamado “projeto de software”. Talvez você tenha ouvido esse termo antes, e talvez tenha até lido alguns livros a respeito. Mas vamos olhar para ele de um modo mais atual, apresentando definições novas e precisas.

Sabemos o que é software. Então, o que realmente temos que definir é a palavra “projeto”:

verbo

1. Elaborar um plano para uma criação. Exemplo: O engenheiro projetará uma ponte neste mês e a construirá no próximo mês.

substantivo

1. Um plano que foi elaborado para uma criação que ainda não foi construída. Exemplo:

O engenheiro surgiu com um projeto para uma ponte; ele a construirá no próximo mês.

2. O plano que uma criação existente segue. Exemplo: Aquela ponte lá segue um bom projeto.

Todas essas definições se aplicam quando falamos sobre projeto de software:

- Quando estamos “projetando software” (“projetar”, como um verbo), estamos planejando-o. Há muitas coisas para planejar sobre software – a estrutura do código, quais tecnologias vamos usar etc. Há muitas decisões técnicas a tomar. Frequentemente, só tomamos essas decisões em nossas mentes, mas, às vezes, também anotamos nossos planos ou desenhamos alguns diagramas.

- Uma vez que fizemos isso, o resultado é um “projeto de software” (“projeto”, como a primeira definição do nome). Esse é o plano que fizemos. Mais uma vez, esse poderia ser um documento escrito, mas também poderia ser apenas várias decisões que tomamos que agora estamos mantendo em nossas mentes.
- Código que já existe também tem “um projeto” (“projeto”, como a segunda definição do nome), que é a estrutura que ele tem ou o plano que ele parece seguir. Algum código pode parecer não ter absolutamente nenhuma estrutura clara – esse código não tem “nenhum projeto”, significando que não houve nenhum plano definido feito pelo programador original. Entre “nenhum projeto” e “um projeto”, também há muitas possibilidades, tais como “um projeto parcial”, “vários projetos conflitantes em um trecho de código”, “um projeto quase completo” etc. Também pode haver projetos efetivamente ruins que são piores do que nenhum projeto. Por exemplo, imagine se deparar com algum código que tenha sido escrito intencionalmente desorganizado ou complexo – esse seria um código com um projeto efetivamente ruim.

A ciência de projeto de software é uma ciência para elaborar planos e tomar decisões sobre software. Ela ajuda as pessoas a tomar decisões como:

- Qual deveria ser a estrutura do código de nosso programa?
- É mais importante focar em ter um programa rápido ou um programa cujo código seja de fácil leitura?
- Para nossas necessidades, qual linguagem de programação deveríamos usar?

Projeto de software não aborda assuntos como:

- Qual deveria ser a estrutura da empresa?
- Quando deveríamos ter reuniões de equipe?
- Quais horas do dia os programadores deveriam trabalhar?
- Como deveríamos avaliar o desempenho de nossos programadores?

Essas não são decisões sobre seu software, são decisões sobre você ou sua organização. Certamente, é importante tomar essas decisões de modo adequado – muitos projetos de software falharam porque não tinham um bom gerenciamento. Mas esse não é o foco deste livro. Este livro é sobre como tomar as decisões técnicas corretas sobre seu software.

Qualquer coisa que envolve a arquitetura de seu sistema de software ou as decisões técnicas que você toma enquanto cria o sistema se enquadra na categoria de “projeto de software”.

Todo programador é um projetista

Todo programador que está trabalhando em um projeto de software está envolvido em projeto. O desenvolvedor líder está encarregado de projetar a arquitetura geral do programa inteiro. Os programadores seniores estão encarregados de projetar suas próprias grandes áreas. E os programadores juniores estão encarregados de projetar suas partes do programa, mesmo que sejam tão simples quanto uma parte de um arquivo. Há até certa quantidade de projeto envolvido mesmo na escrita de uma única linha de código.

Mesmo quando você está programando tudo sozinho, ainda há um processo de projeto que continua. Às vezes, você toma uma decisão imediatamente antes de os seus dedos pressionarem o teclado, e esse é todo o processo. Outras vezes, você pensa sobre como vai escrever o programa quando estiver na cama à noite.

Todos aqueles que escrevem software são projetistas. Cada pessoa em uma equipe de software é responsável por se certificar de que seu código está bem projetado. Ninguém que esteja escrevendo código para um projeto de software pode ignorar projeto de software, em qualquer nível.

Contudo, isso não significa que o projeto é uma democracia. Você não deve projetar em conjunto – o resultado será um projeto efetivamente ruim. Em vez disso, todos os desenvolvedores devem ter a autoridade de tomar boas decisões de projeto em suas próprias áreas. Se eles tomarem decisões erradas ou medíocres, essas devem ser canceladas por um desenvolvedor sênior ou pelo programador líder, os quais devem ter

poder de veto sobre os projetistas a eles subordinados.¹ Mas, de outro modo, a responsabilidade pelo projeto do código deve permanecer com as pessoas que estão realmente trabalhando nele.

Um projetista deve sempre estar disposto a ouvir sugestões e feedback, porque os programadores normalmente são pessoas inteligentes que têm boas ideias. Mas após considerar todos os detalhes, qualquer decisão dada deve ser tomada por uma pessoa, não por um grupo de pessoas.

A ciência de projeto de software

O projeto de software, como é praticado no mundo hoje, não é uma ciência. O que é uma ciência? A definição do dicionário é um pouco complexa, mas, basicamente, para que um assunto seja uma ciência, ele tem que passar por certos testes:

- Uma ciência deve conter conhecimento que foi coletado. Isto é, ela tem que ser composta de fatos – não de opiniões – e esses fatos devem ter sido reunidos em algum lugar (como em um livro).
- Esse conhecimento deve ter algum tipo de organização. Tem que ser dividido em categorias, as diversas partes devem estar corretamente relacionadas umas às outras em termos de importância etc.
- Uma ciência deve conter verdades gerais ou leis básicas.
- Uma ciência deve lhe dizer como fazer algo no universo físico. Ela dever ser de algum modo aplicável no trabalho ou na vida.
- Normalmente, uma ciência é descoberta e comprovada por meio de método científico, o que envolve observação do universo físico, elaborar uma teoria sobre como o universo funciona, realizar experimentos para verificar sua teoria e mostrar que o mesmo experimento funciona em todo lugar para demonstrar que a teoria é uma verdade geral e não apenas uma coincidência ou algo que funcionou só para você.

¹ Se você for a pessoa que estiver cancelando uma decisão, tente instruir o outro programador quando fizer isso. Mostre como ou por que sua decisão é melhor que a dele. Se fizer isso, com o tempo você terá que contestar aquele programador cada vez menos. Alguns programadores nunca aprendem, contudo – se, após vários meses ou anos de tal instrução, um programador continuar a tomar inúmeras decisões erradas, ele deve ser afastado de sua equipe. Contudo, a maior parte dos programadores são pessoas muito espertas que percebem as coisas rapidamente, então isso raramente será uma preocupação.

No mundo do software, temos muito conhecimento; ele foi coletado em livros e foi um tanto organizado. Contudo, de todas as partes exigidas para se fazer uma ciência, estamos sentindo falta da parte mais importante: leis claramente declaradas – verdades inabaláveis que nunca falharão.

Desenvolvedores de software experientes sabem qual é a coisa certa a fazer, mas por que aquela é a coisa certa? O que torna algumas decisões certas e algumas decisões erradas?

Quais são as leis fundamentais do projeto de software?

Este livro é o registro de sua descoberta. Ele expõe um conjunto de definições, fatos, regras e leis para desenvolvimento de software, em sua maioria focados em projeto de software. Qual é a diferença entre um fato, uma regra, uma definição e uma lei?

- *Definições* lhe dizem o que algo é e como você o usaria.
- *Fatos* são apenas declarações verdadeiras sobre algo. Qualquer fragmento de informação verdadeira é um fato.
- *Regras* são declarações que lhe dão conselho verdadeiro, abrangem algo específico e ajudam a orientar decisões, mas não o ajudam necessariamente a prever o que acontecerá no futuro ou a descobrir outras verdades. Elas normalmente lhe dizem se deve ou não tomar alguma decisão.
- *Leis* são fatos que serão sempre verdadeiros e abrangem uma ampla área do conhecimento. Elas ajudam a descobrir outras verdades importantes e permitem que você preveja o que acontecerá no futuro.

De todas essas, as leis são as mais importantes. Neste livro, você saberá que algo é uma lei porque o texto dirá isso explicitamente. Se você não estiver certo sobre em qual categoria alguma informação se enquadra, o apêndice B lista toda informação importante no livro e a rotula claramente como uma lei, regra, definição ou apenas um simples e antigo fato.

Quando você ler algumas dessas definições, leis, regras ou fatos, poderá dizer a você mesmo: “Isso era óbvio, eu já sabia disso”. Isso é realmente bastante esperado – se você está na área de desenvolvimento de software há muito tempo, possivelmente se deparou com algumas dessas ideias antes. Contudo, quando tiver essa reação, pergunte a você mesmo:

- Eu sabia que esse dado particular era comprovadamente verdadeiro?
- Eu sabia quão importante ele era?
- Eu poderia tê-lo comunicado claramente a outra pessoa para que ela o entendesse completamente?
- Eu entendi como ele se relacionava com outros dados na área de desenvolvimento de software?

Se puder dizer “sim” a algumas dessas perguntas quando anteriormente teria dito “não” ou “talvez”, então você obteve um tipo específico de entendimento, e esse entendimento é uma parte enorme do que diferencia uma ciência de um mero conjunto de ideias.

Claro, essa ciência pode não ser perfeita ainda. Há sempre algo mais a descobrir no universo, mais a saber sobre qualquer área. Às vezes, há até correções a serem feitas em leis básicas quando dados novos são descobertos ou surgem fatos novos. Mas esse é um ponto de partida! É algo sobre o qual se pode construir: um conjunto real de leis e verdades observáveis para criar software.

Mesmo que partes deste livro se comprovem erradas algum dia e uma ciência melhor seja desenvolvida, é importante que um fato permaneça claro: projeto de software pode ser uma ciência. Esse não é um mistério eterno, sujeito à opinião de todo programador que aparece ou todo consultor que quer ganhar algum dinheiro vendendo seu “novo método” de projeto de software:

Há leis, elas podem ser conhecidas e você pode conhecê-las. Elas são eternas, imutáveis e fundamentalmente verdadeiras e funcionam.

Quanto ao fato de as leis neste livro serem leis corretas ou não... bem, há centenas de exemplos e experimentos que poderiam ser citados para comprová-las, mas, afinal, você terá que decidir por você mesmo. Teste as leis. Veja se consegue pensar em quaisquer verdades gerais sobre desenvolvimento de software que sejam mais amplas ou mais fundamentais. E se você surgir com qualquer coisa ou encontrar quaisquer problemas com as leis, veja <http://www.codesimplicity.com/> para contatar o autor com suas contribuições ou perguntas. Quaisquer revelações adicionais a este assunto beneficiarão todo mundo, desde que essas revelações sejam realmente verdadeiras, leis fundamentais ou regras do projeto de software.

Por que não houve nenhuma ciência de projeto de software?

Talvez você esteja interessado em saber por que uma ciência para projeto de software não existia antes deste livro. Afinal de contas, já estamos escrevendo software há décadas e décadas.

Bem, essa é uma história interessante. Aqui estão alguns acontecimentos que podem ajudar você a entender como chegamos tão longe com os computadores sem desenvolver uma ciência de projeto de software.

O que pensamos hoje serem “computadores” começou nas mentes dos matemáticos como dispositivos puramente abstratos – pensamentos sobre como solucionar problemas matemáticos usando máquinas em vez do cérebro.

Esses matemáticos são as pessoas que consideraríamos os fundadores da ciência da computação, que é o estudo matemático do processamento da informação. Não é, como algumas pessoas acreditam, o estudo da programação de computadores.

Contudo, os primeiríssimos computadores foram construídos sob a supervisão desses cientistas da computação por engenheiros eletrônicos altamente qualificados. Eles eram operados por pessoas altamente treinadas em ambientes rigorosamente controlados. Eles foram todos construídos sob medida pelas organizações que precisavam deles (a maior parte governos, para apontar mísseis e decifrar códigos), e havia apenas uma ou duas cópias de qualquer dado modelo.

Então surgiu o UNIVAC e os primeiros computadores comerciais do mundo. Nesse momento, havia apenas alguns matemáticos teóricos avançados no mundo. Quando os computadores começaram a se tornar disponíveis a todo mundo, claramente não era possível enviar um matemático junto com um computador. Então, embora algumas organizações, tais como o Escritório do Censo dos Estados Unidos (um dos primeiros a receber um UNIVAC), quase certamente tivessem alguns operadores altamente treinados para seu equipamento, outras organizações sem dúvida receberam suas máquinas e disseram: “OK, Bill da Contabilidade, esse é seu! Leia o manual e se vire!”. E lá foi Bill, mergulhando nesta máquina complexa e fazendo o possível para fazê-la funcionar.

Bill tornou-se assim um dos primeiros “programadores práticos”. Talvez ele tivesse estudado matemática na escola, mas quase certamente não estudara o tipo de teoria avançada necessária para conceber e projetar a máquina em si. Contudo, ele podia ler o manual e entendê-lo e, por tentativa e erro, fazer a máquina executar o que ele queria.

Claro, quanto mais computadores comerciais eram despachados, mais Bills e menos operadores altamente treinados tínhamos. A grande maioria dos programadores terminou exatamente como Bill. E se havia uma coisa de que Bill sofria, era de pressão no trabalho. Ele recebia pedidos da administração para “Fazer aquela tarefa agora!” e ouvia: “Não nos preocupamos como é feita, apenas a faça!”. Ele entendeu como fazer a coisa funcionar de acordo com seu manual, e ela realmente funcionou, embora quebrasse de duas em duas horas.

Posteriormente Bill conseguiu toda uma equipe de programadores para trabalhar com ele. Ele teve que entender como projetar um sistema e dividir as tarefas entre pessoas diferentes. Toda a arte da programação prática cresceu organicamente, mas como estudantes universitários aprendendo a cozinhar sozinhos do que como os engenheiros da NASA construindo o ônibus especial.

Então, nesse estágio, há um sistema confuso de desenvolvimento de software, e é tudo muito complexo e difícil de gerenciar, mas todo mundo se vira de algum modo. Então surgiu *The Mythical Man Month - O Mítico Homem-Mês* (Addison-Wesley), um livro de Fred Brooks, que realmente olhou para o processo de desenvolvimento de software em um projeto real e apontou alguns fatos sobre ele – o mais famoso, segundo o qual incluir programadores em um projeto de software atrasado atrasa-o ainda mais. Ele não apareceu com uma ciência completa, mas realmente fez algumas boas observações sobre programação e gerenciamento de desenvolvimento de software.

Depois disso veio uma variedade de métodos de desenvolvimento de software: o Processo Racional Unificado, o Modelo de Maturidade de Capacidade, Desenvolvimento Ágil de Software e muitos outros. Nenhum desses reivindicou ser uma ciência – eram apenas modos de gerenciar a complexidade do desenvolvimento de software.

E isso, basicamente, nos traz até onde estamos hoje: muitos métodos, mas nenhuma ciência verdadeira.

De fato, há duas ciências ausentes: a ciência de gerenciamento de software e a ciência de projeto de software.

A ciência de gerenciamento de software nos diria como dividir trabalho entre programadores, como agendar lançamentos, como estimar a quantidade de tempo que um trabalho levará e assim por diante. Ela está sendo trabalhada ativamente e é tratada pelos diversos métodos mencionados. O fato de opiniões conflitantes e igualmente válidas existirem dentro da área indica que as leis fundamentais do gerenciamento de software ainda não foram elaboradas. Contudo, há pelo menos atenção sendo dada ao problema.

A ciência de projeto de software, por outro lado, recebe pouca atenção no mundo prático da programação. Poucas pessoas são ensinadas na escola que poderia haver uma ciência no projeto de software. Em vez disso, no geral, elas ouvem: “É assim que essa linguagem de programação funciona; agora vá escrever algum software!”.

Este livro existe para preencher essa lacuna.

A ciência apresentada aqui não é ciência da computação. Isso é estudo matemático. Em vez disso, este livro contém os primórdios de uma ciência para o “programador prático” – um conjunto de leis e regras fundamentais a seguir quando da escrita de um programa em qualquer linguagem. Essa é uma ciência que é tão confiável quanto Física ou Química ao lhe dizer como criar uma aplicação.

Disseram que tal ciência não é possível; que o projeto de software é variável demais para ser definido por leis simples e fundamentais; e que tudo é apenas uma questão de opinião. Algumas pessoas também disseram certa vez que entender o universo físico era impossível porque “é a criação de Deus, e Deus é incognoscível”, e, contudo, descobrimos ciências para o universo físico. Então, a menos que você acredite que os computadores sejam incognoscíveis, criar uma ciência de projeto de software deve ser inteiramente possível.

Há também um mito comum de que programação é inteiramente uma forma de arte, totalmente sujeita a desejos pessoais de um programador individual. Contudo, embora seja verdade que há uma boa parte de arte envolvida na aplicação de qualquer ciência, ainda tem que haver uma ciência lá para aplicar, e atualmente não há nenhuma.

A fonte primária de complexidade em software é, de fato, a falta dessa ciência. Se os programadores realmente tivessem uma ciência para simplicidade em software, não haveria tanta complexidade e não precisaríamos de processos malucos para gerenciar essa complexidade.

CAPÍTULO 3

As forças motoras do projeto de software

Quando escrevemos software, devemos ter alguma ideia do por que estamos fazendo isso e qual é o objetivo final.

Há algum modo pelo qual poderíamos resumir o propósito de todo software? Se tal declaração fosse possível, ela daria orientação a toda a nossa ciência de projeto de software, porque saberíamos o que estávamos buscando.

Bem, há, de fato, um único propósito para todo software:

Ajudar as pessoas.¹

Podemos dividir isso em um propósito mais específico para partes individuais de softwares. Por exemplo, um processador de texto existe para ajudar as pessoas a escrever coisas, e um navegador web existe para ajudar as pessoas a navegar na Web.

Alguns softwares existem apenas para ajudar grupos específicos de pessoas. Por exemplo, há muitos softwares contábeis que existem para ajudar os contadores; esses visam apenas aquele grupo específico de pessoas.

E os softwares que ajudam animais ou plantas? Bem, seu propósito é realmente ajudar as pessoas a ajudar animais ou plantas.

O importante aqui é que o software nunca está lá para ajudar objetos inanimados. O software não existe para ajudar o computador, ele sempre existe para ajudar as pessoas. Mesmo quando você está escrevendo

¹ Esse fato (o propósito de todo software) é mais importante do que uma lei. Não há uma palavra simples para esse tipo de fato, em inglês. Poderíamos talvez chamá-lo de “lei antiga,” embora não preencha inteiramente os critérios para uma lei (por exemplo, não prevê o futuro). Por razões de simplicidade, os apêndices no final deste livro listam este fato como uma lei, e de outro modo apenas nos referimos a ele como “o Propósito do Software”.

bibliotecas, você está escrevendo para ajudar os programadores, que são pessoas. Você nunca está escrevendo para ajudar o computador.

Agora, o que significa “ajudar”? De algumas maneiras, isso é subjetivo – o que ajuda uma pessoa pode não ajudar outra. Mas a palavra tem uma definição de dicionário, então não depende completamente de cada pessoa o que a palavra significa. O Webster’s New World Dictionary of the American Language define “ajudar” como:

Tornar mais fácil para (uma pessoa) fazer algo; auxiliar; assistir. Especificamente... fazer parte do trabalho de; facilitar ou compartilhar o trabalho de.

Há muitas coisas nas quais você poderia ajudar – organizar um horário, escrever um livro, planejar uma dieta, qualquer coisa. No que você ajuda depende de você, mas o propósito é sempre ajudar.

O propósito do software não é “ganhar dinheiro” ou “exibir como sou inteligente”. Qualquer pessoa que escreva um software tendo esses como seus únicos propósitos está violando o propósito do software e é muito provável que terá problemas. Tudo bem, esses são meios de “ajudar” a você mesmo, mas esse é um escopo muito limitado de ajuda, e projetar com apenas esses propósitos em mente provavelmente levará a um software de qualidade inferior do que genuinamente projetar para ajudar as pessoas a fazer o que precisam ou querem fazer.²

Pessoas que não conseguem conceber ajudar outra pessoa escreverão software ruim – isto é, o software delas não ajudará muito as pessoas. De fato, poderia ser teorizado (como uma suposição, baseada na observação de muitos programadores no decorrer do tempo) que a sua capacidade potencial de escrever bom software é limitada apenas por sua capacidade de conceber ajudar o outro.

Em geral, quando estamos tomando decisões sobre software, nosso princípio orientador pode ser como podemos ajudar. (E lembre-se, há graus

² Observe que “ganhar dinheiro” pode ser certamente um de seus propósitos pessoais ou um propósito de sua organização – não há nada de errado em ganhar dinheiro. Só não deveria ser o propósito de seu software. Em qualquer caso, a quantia de dinheiro que você ganha provavelmente está diretamente relacionada com quanto seu software ajuda as pessoas. De fato, os dois fatores primordiais que determinam a receita de uma empresa de software são provavelmente a habilidade comercial de sua organização (incluindo administração, gerência, marketing e vendas) e quanto seu software ajuda as pessoas.

variados de ajuda – pode-se ajudar muito ou um pouco, muitas pessoas ou apenas algumas). Você pode até priorizar solicitações de recursos desse jeito. Qual recurso ajudará mais as pessoas? Este recurso deve receber a mais alta prioridade. Há mais a saber sobre priorizar recursos, mas “Quanto isso ajuda nossos usuários?” é uma pergunta boa e básica a ser feita sobre qualquer alteração proposta em seu sistema de software.

Geralmente, este propósito – ajudar pessoas – é o mais importante que se deve ter em mente quando se projeta software, e defini-lo nos permite agora criar e entender uma verdadeira ciência do projeto de software.

Aplicação no mundo real

Como podemos aplicar o propósito do software a nossos projetos no mundo real? Bem, digamos que estamos escrevendo um editor de texto para programadores. A primeira coisa que precisamos fazer é determinar o propósito de nosso software. É melhor mantê-lo simples, digamos que o propósito é “ajudar programadores a editar texto”. É bom ser mais específico do que isso, e às vezes é útil, mas se o grupo não concorda sobre um propósito específico, que pelo menos venha com um simples assim.

Agora que temos um propósito, vamos olhar para todas as nossas solicitações de recursos. Para cada uma podemos nos perguntar: “Como esse recurso ajudaria os programadores a editar texto?” Se a resposta for: “Não ajudaria”, podemos imediatamente excluir esse recurso de nossa lista. Então, para cada um dos recursos remanescentes, podemos anotar a resposta como uma frase curta. Por exemplo, suponha que alguém nos peça para adicionar atalhos de teclado para ações comuns. Poderíamos dizer: “Isso ajuda os programadores a editar texto porque permite que eles interajam com o programa mais rapidamente sem fazer uma longa pausa para digitação”. (Na verdade, você não tem que anotar essas coisas se isso não parece prático para sua situação – apenas ter alguma ideia da resposta para você mesmo é suficiente).

Também há várias outras razões úteis para fazer essa pergunta:

- Ajuda a resolver incertezas sobre a descrição do recurso ou como ele deve ser implementado. Por exemplo, a resposta anterior sobre atalhos de teclado nos diz que a implementação deve ser rápida, porque esse é o valor que os usuários extraem dela.

- Ajuda a equipe a chegar a um acordo sobre o valor de um recurso. Algumas pessoas podem não gostar da ideia de atalhos de teclado, mas todo mundo deve poder concordar que a resposta anterior explica por que eles são valiosos. De fato, alguns desenvolvedores podem até ter uma ideia melhor de como satisfazer a necessidade desse usuário (interagindo com o editor de texto mais rapidamente) sem atalhos de teclado. Isso é bom! Se a resposta nos levar a uma ideia melhor de recurso, devemos implementar essa em vez daquela. A resposta nos diz o que é realmente necessário, não só o que o usuário pensou que ele queria.
- Responder à pergunta tornará óbvio que alguns recursos são mais importantes do que outros. Isso ajuda os líderes de projetos a priorizar o trabalho.
- Na pior das hipóteses, se nosso editor de texto tiver se tornado grande com recursos demais no decorrer do tempo, a resposta poderá nos ajudar a decidir quais recursos devem ser eliminados.

Poderíamos também fazer uma lista de erros, que poderíamos examinar e fazer a pergunta contrária: “Como esse erro atrapalha a edição de texto pelos programadores?”. Às vezes a resposta é óbvia, então, ela realmente não precisa ser anotada. Por exemplo, se o programa falha quando você tenta salvar um arquivo, você não precisa explicar por que isso é ruim.

Provavelmente há inúmeros outros modos de aplicar o propósito do software no trabalho diário; esses são apenas alguns exemplos.

Os objetivos do projeto de software

Agora que conhecemos o propósito do software, podemos dar um pouco de direcionamento à nossa ciência do projeto de software.

A partir do propósito, sabemos que, quando escrevemos software, estamos tentando ajudar as pessoas. Então, um dos objetivos de uma ciência de projeto de software deve ser:

Permitir que escrevamos software que seja o mais útil possível.

Em segundo lugar, normalmente queremos que as pessoas continuem sendo ajudadas por nosso software. Então, nosso segundo objetivo é:

Permitir que nosso software continue a ser o mais útil possível.

Agora, esse é um ótimo objetivo, mas qualquer sistema de software, de qualquer tamanho, é extremamente complexo, então, permitir que ele continue a ser útil no decorrer do tempo é uma tarefa e tanto. De fato, a principal barreira hoje para escrever e manter um software útil é a verdadeira dificuldade de projeto e programação. Quando o software é difícil de ser criado ou alterado, os programadores passam a maior parte do tempo focados em fazer as coisas “simplesmente funcionar”, e menos tempo focados em ajudar o usuário. Mas, quando um sistema é fácil de se trabalhar, os programadores podem passar mais tempo focados em ser úteis ao usuário e menos tempo focados nos detalhes de programação. De modo semelhante, quanto mais fácil é fazer a manutenção de um software, mais fácil é para os programadores assegurarem que o software continue a ser útil.

Isso nos leva ao nosso terceiro objetivo:

Projetar sistemas que possam ser criados e submetidos à manutenção o mais facilmente possível por seus programadores, para que eles possam ser – e continuem a ser – o mais úteis possível.

Esse terceiro objetivo é aquele tradicionalmente considerado o objetivo do projeto de software, mesmo que isso nunca seja declarado explicitamente. Contudo, é muito importante também ter o primeiro e segundo objetivos para nos guiar. Queremos lembrar que ser útil agora e no futuro são as motivações para esse terceiro objetivo.

Uma coisa que é importante assinalar sobre esse terceiro objetivo é a frase “o mais facilmente possível”. A ideia é tornar nossos programas fáceis de criar e de submeter à manutenção, não torná-los difíceis ou complexos. Isso não significa que tudo será imediatamente fácil – às vezes leva tempo aprender uma nova tecnologia ou projetar algo bem –, mas, no longo prazo, suas escolhas devem tornar a criação e manutenção de seu software mais fáceis.

Às vezes, o primeiro objetivo (ser útil) e o terceiro objetivo (facilidade de manutenção) estão um pouco em conflito – tornar seu software útil pode torná-lo mais difícil de submeter à manutenção. Contudo, esses dois objetivos têm estado historicamente muito mais em conflito do que necessitariam estar. É absolutamente possível criar um sistema totalmente sujeito à manutenção que seja extremamente útil a seus usuários. E, de fato, se você não o tornar sujeito à manutenção, vai ser bastante difícil satisfazer o segundo objetivo de continuar a ser útil. Então, o terceiro objetivo é importante porque os dois primeiros não podem ser alcançados de outro modo.

CAPÍTULO 4

O futuro

A pergunta primordial enfrentada pelos projetistas de software é: “Como tomo decisões sobre meu software?”. Quando defrontado com muitas direções possíveis nas quais você poderia ir, qual opção é a melhor? Nunca é uma questão de qual decisão seria absolutamente certa *versus* qual decisão seria absolutamente errada. Em vez disso, o que queremos saber é: “Dadas muitas decisões possíveis, quais dessas são melhores que outras?”. É uma questão de estabelecer prioridades entre as decisões e então escolher a melhor dentre todas as possibilidades. Por exemplo, um projetista pode se perguntar: “Há 100 recursos diferentes nos quais poderíamos trabalhar hoje, mas temos apenas mão de obra para trabalhar em dois. Em quais devemos trabalhar primeiro?”

A equação do projeto de software

A pergunta anterior, e de fato toda pergunta dessa natureza em projeto de software, é respondida por esta equação:

$$D = \frac{V}{E}$$

onde:

D representa a *desejabilidade* por uma alteração. O quanto queremos fazer algo?

V representa o *valor* de uma alteração. Quão valiosa é essa alteração? Normalmente, você determinaria isso perguntando: “Quanto isso ajuda nossos usuários?”, embora haja outros métodos para determinar valor também.

E representa o *esforço* envolvido na realização da alteração. Quanto trabalho a alteração exigirá?

Essencialmente, essa equação diz:

A desejabilidade por qualquer alteração é diretamente proporcional ao valor da alteração e inversamente proporcional ao esforço envolvido em realizar a alteração.

Elas não diz se uma alteração é absolutamente certa ou errada; em vez disso, ela lhe diz para estabelecer prioridades entre suas opções. Alterações que trarão muito valor e exigem pouco esforço são “melhores” do que aquelas que trarão pouco valor e exigem muito esforço.

Mesmo que sua pergunta seja: “Devemos permanecer os mesmos e não mudar?”, essa equação lhe dá a resposta. Pergunte-se: “Qual é o valor de permanecer o mesmo?” e “Qual é o esforço envolvido em permanecer o mesmo?” e compare isso ao valor de mudar e ao esforço envolvido na alteração.

Valor

O que queremos dizer com “valor” na equação? A definição mais simples de valor seria:

O grau em que essa alteração ajuda qualquer pessoa em qualquer lugar.

As pessoas mais importantes a ajudar são seus usuários. Contudo, desenvolver recursos (features) que o ajudarão a se sustentar financeiramente também é uma forma de valor – é valioso para você. De fato, há muitos modos pelos quais uma alteração pode ter valor; esses são apenas dois exemplos.

Às vezes, determinar o valor numérico real e preciso de qualquer alteração particular é difícil. Por exemplo, digamos que seu software ajuda as pessoas a perder peso. Como você determina o valor exato de ajudar alguém a perder peso? Você não pode, realmente. Mas você pode saber com precisão que alguns recursos do software ajudarão as pessoas a perder bastante peso e alguns recursos não ajudarão absolutamente as pessoas a perder. Então, você ainda pode estabelecer prioridades entre as alterações pelo seu valor.

Entender o valor de cada possível alteração vem sobretudo da experiência como desenvolvedor e de fazer pesquisa adequada com usuários para descobrir o que mais os ajudará.

Probabilidade do valor e valor potencial

Valor é realmente composto de dois fatores: a probabilidade do valor (quão provável é que essa alteração ajudará um usuário), e o valor potencial (quanto essa alteração ajudará um usuário durante aqueles momentos em que ela realmente ajuda essa pessoa).

Por exemplo:

- Um recurso que poderia salvar a vida de alguém, mesmo que haja apenas uma chance em um milhão de ele ser necessário, ainda é um recurso altamente valioso. Ele tem um alto valor potencial (salvar uma vida), embora tenha baixa probabilidade de valor.

Como outro exemplo, em um programa de planilha eletrônica você poderia adicionar um recurso que ajudasse pessoas cegas a digitar números no sistema. Apenas uma pequena porcentagem das pessoas é cega, mas sem esse recurso elas não poderiam absolutamente usar seu software. Novamente, esse recurso é valioso porque tem um valor potencial muito alto, apesar de afetar apenas um pequeno grupo de usuários (uma baixa probabilidade de valor).

- Se há um recurso que fará 100% de seus usuários sorrir, esse também é um recurso valioso. Ele tem um valor potencial muito pequeno (fazer as pessoas sorrir), mas afeta um número muito grande de usuários, então ele tem uma alta probabilidade de valor.
- Por outro lado, se você implementar um recurso que tem apenas uma chance em um milhão de fazer alguém sorrir, isso não é muito valioso. Esse é um recurso com baixo valor potencial e baixa probabilidade de valor.

Então, quando considerar o valor, você também tem que considerar:

- Para quantos usuários (qual porcentagem) essa mudança será valiosa?

- Qual é a probabilidade de que esse recurso seja valioso para um usuário? Ou, colocado de outra forma: com que frequência esse recurso será valioso?
- Quando é valioso, quão valioso será?

Equilíbrio de danos

Algumas alterações podem causar algum dano além da ajuda que trazem. Por exemplo, alguns usuários podem ficar aborrecidos se seu software mostrar-lhes anúncios, mesmo que esses anúncios ajudem a sustentar você como desenvolvedor.

Calcular o valor de uma alteração inclui considerar quanto dano ela pode causar e equilibrar isso com a ajuda que ela traz.

O valor de ter usuários

Recursos (features) que não têm usuários não têm valor imediato. Isso pode incluir recursos que os usuários não conseguem encontrar, recursos que são difíceis demais de usar ou recursos que simplesmente não ajudam ninguém. Eles poderão ter valor no futuro, mas não têm valor agora.

Isso também que dizer que, na maioria dos casos, você deve realmente liberar seu software a fim de que ele seja valioso. Uma alteração que leva muito tempo para ser feita pode realmente acabar tendo valor zero, porque ela não foi liberada a tempo de ajudar as pessoas de modo eficaz. Pode ser importante levar em conta horários de execução quando determinar a desejabilidade por alterações.

Esforço

Esforço é um pouco mais fácil de colocar em números do que valor. Normalmente, você pode descrever esforço como “certo número de horas de trabalho por certo número de pessoas”. “Cem anos-pessoa” é um exemplo de uma medição numérica normalmente ouvida para esforço, representando 100 anos de trabalho por 1 pessoa, 1 ano de trabalho por 100 pessoas, 2 anos de trabalho por 50 pessoas etc.

Contudo, embora o esforço possa ser colocado em números, medi-lo em situações práticas é muito complicado – talvez impossível. Alterações podem ter muitos custos não aparentes que podem ser difíceis de prever, tais como o tempo que você levará no futuro corrigindo quaisquer erros causados pelas alterações. Mas se você for um desenvolvedor de software experiente, ainda pode estabelecer prioridades entre as alterações pelo quanto de esforço elas provavelmente exigirão, mesmo que você não saiba os números exatos para cada uma.

Quando considerar o esforço envolvido em uma alteração, é importante levar em conta todo o esforço que poderia estar envolvido, não apenas o tempo que você vai passar programando. Quanta pesquisa será necessária? Quanta comunicação todos os desenvolvedores terão que fazer uns com os outros? Quanto tempo você passará pensando na alteração?

Em suma, cada única parcela de tempo associada a uma alteração é parte do custo do esforço.

Manutenção

A equação como a temos até agora é muito simples, mas está faltando um elemento importante – tempo. Não só você tem que implementar uma alteração, mas você também tem que submetê-la à manutenção no decorrer do tempo. Todas as alterações exigem manutenção. Isso é muito óbvio com algumas alterações – se estiver escrevendo um programa para fazer os impostos das pessoas, você vai ter que atualizá-lo para as novas leis tributárias todos os anos. Mas mesmo alterações que não parecem imediatamente ter um custo de manutenção de longo prazo terão um, mesmo que seja apenas o custo pela necessidade de se certificar de que o código ainda funciona quando você o estiver testando no próximo ano.

Devemos também considerar o valor tanto agora quanto no futuro. Quando implementamos alguma alteração em nosso sistema, ela ajudará nossos usuários atuais, mas poderá também ajudar todos os nossos futuros usuários. Poderá até afetar o número total de futuros usuários, alterando, assim, o quanto nosso software como um todo ajuda as pessoas.

Alguns recursos chegam a mudar de valor com o tempo. Por exemplo, elaborar um programa que calcule impostos de acordo com as leis tributárias do ano de 2009 é valioso em 2009 e 2010, mas não tão valioso quando 2011 se aproximasse. Esse é um recurso que se torna menos valioso com o tempo. Alguns recursos também se tornam mais valiosos com o decorrer do tempo.

Então, olhando para isso de modo realista, vemos que o esforço realmente envolve tanto o esforço de implementação quanto o esforço de manutenção, e o valor envolve tanto o valor agora quanto o valor no futuro. Em forma de equação, isso parece assim:

$$E = E_i + E_m$$

$$V = V_n + V_f$$

onde:

E_i representa o *esforço de implementação*.

E_m representa o *esforço de manutenção*.

V_n representa o *valor agora*.

V_f representa o *valor futuro*.

A equação completa

Com tudo conectado, a equação completa parece assim:

$$D = \frac{V_n + V_f}{E_i + E_m}$$

Ou, em português:

A desejabilidade por uma alteração é diretamente proporcional ao valor agora mais o valor futuro, e inversamente proporcional ao esforço de implementação mais o esforço de manutenção.

Essa é a lei principal do projeto de software. Contudo, há um pouco mais a saber sobre ela.

Reduzindo a equação

“Valor futuro” e “esforço de manutenção” dependem ambos do tempo, o que faz com que coisas interessantes aconteçam com a equação quando a aplicamos a uma situação do mundo real. Para demonstrá-las, vamos fingir que podemos usar dinheiro para solucionar a equação tanto para valor quanto para esforço. “Valor” será medido por quanto dinheiro obteremos com a alteração. “Esforço” será medido em termos de quanto dinheiro nos custará implementar a alteração. Você não deve usar a equação desse jeito no mundo real, mas, por causa de nosso exemplo, isso vai simplificar as coisas.

Então, digamos que temos uma alteração que queremos fazer, onde a equação parece assim:

$$D = \frac{\$10,000 + \$100/\text{dia}}{\$1,000 + \$100/\text{dia}}$$

Em outras palavras, essa alteração custa \$1.000 para implementar (esforço de implementação, inferior à esquerda) e nos dá \$10.000 imediatamente (valor agora, superior à esquerda). Então, cada dia depois disso, ela nos dá \$1.000 (valor futuro, superior à direita) e ela custa \$100 para fazer manutenção (esforço de manutenção, inferior à direita).

Após 10 dias, o valor futuro acumulado totaliza \$10.000 e o esforço de manutenção totaliza \$1.000. Isso é igual aos originais “valor agora” e custo de implementação, após apenas 10 dias.

Após 100 dias, o valor futuro totaliza \$100.000, e o esforço de manutenção atinge \$10.000.

Após 1000 dias, o valor futuro total atinge \$1.000.000 e o esforço de manutenção totaliza \$100.000. Nesse momento, os originais “valor agora” e custo de implementação parecem muito pequenos em comparação. Com o passar do tempo, eles se tornarão ainda menos significativos, eventualmente perdendo inteiramente a importância. Assim, com o passar do tempo, nossa equação se reduz a isso:¹

¹ Observação opcional para matemáticos: se você estudou cálculo, talvez tenha percebido que estamos começando a analisar o limite da equação enquanto o tempo se aproxima do infinito. Em geral, você deveria estar pensando na Equação do Projeto de Software como se ela fosse uma série infinita com um limite, não apenas uma equação estática. Contudo, em razão da simplicidade, ela está escrita aqui como uma equação estática.

$$D = \frac{V_f}{E_m}$$

E, de fato, quase todas as decisões em projeto de software se reduzem inteiramente em avaliar o valor futuro de uma alteração *versus* seu esforço de manutenção. Há situações em que o valor presente e o esforço de implementação são suficientemente grandes para serem significativos em uma decisão, mas elas são extremamente raras. Em geral, sistemas de software são submetidos à manutenção por tanto tempo que o valor agora e o esforço de implementação têm a garantia de se tornar insignificantes em quase todos os casos quando comparados com o valor futuro e o esforço de manutenção de longo prazo.

O que você quer e o que não quer

A lição primordial a aprender aqui é que queremos evitar situações em que, para uma dada alteração, o esforço de manutenção acabará superando o valor futuro. Por exemplo, imagine que você implementa uma alteração em que o esforço e o valor parecem assim no decorrer de cinco dias:

Dia	Esforço	Valor
1	\$10	\$1.000
2	\$100	\$100
3	\$1.000	\$10
4	\$10.000	\$1
5	\$100.000	\$0,10
Total	\$111.110	\$1.111,10

Claramente, essa é uma terrível, terrível alteração que você jamais deveria ter feito. Se as coisas continuarem nesse ritmo, você não poderá absolutamente fazer a manutenção do sistema – ficará infinitamente caro e o valor que você está ganhando a cada dia se tornará \$0.

Qualquer situação na qual o esforço de manutenção aumenta mais rápido do que o valor vai causar problemas a você, mesmo que pareça bom a princípio:

Dia	Esforço	Valor
1	\$1.000	\$1.000
2	\$2.000	\$2.000
3	\$4.000	\$3.000
4	\$8.000	\$4.000
Total	\$15.000	\$10.000

A solução ideal – e o único modo de garantir sucesso – é projetar seus sistemas de tal forma que o esforço de manutenção *diminua* com o tempo, e eventualmente se torne zero (ou o mais próximo disso possível). Desde que você possa fazer isso, não importa quão grande ou pequeno o valor futuro se torna; você não tem que se preocupar com isso. Por exemplo, essas tabelas mostram situações desejáveis:

Dia	Esforço	Valor
1	\$1.000	\$0
2	\$100	\$10
3	\$10	\$100
4	\$0	\$1.000
5	\$0	\$10.000
Total	\$1.110	\$11.110

Dia	Esforço	Valor
1	\$20	\$10
2	\$10	\$10
3	\$5	\$10
4	\$1	\$10
5	\$0	\$10
Total	\$36	\$50

Alterações com um valor futuro mais alto ainda são mais desejáveis, mas, já que toda decisão tem um custo de manutenção que se aproxima de zero com o tempo, você não pode se colocar em uma situação futura perigosa.

Teoricamente, já que o valor futuro é sempre maior do que o esforço de manutenção, a alteração ainda é desejável. Então, você poderia fazer alguma alteração em que tanto o esforço de manutenção quanto o valor futuro aumentasse, desde que o valor futuro continuasse sendo suficientemente grande para superar o esforço de manutenção:

Dia	Esforço	Valor
1	\$1	\$0
2	\$2	\$2
3	\$3	\$4
4	\$4	\$6
5	\$5	\$8
Total	\$15	\$20

Tal alteração não é ruim, mas é mais desejável fazer uma alteração cujo esforço de manutenção diminua, mesmo que ela tenha um esforço de implementação maior. Se o esforço de manutenção diminui, a alteração realmente se torna cada vez mais desejável com o tempo. Isso a torna uma escolha melhor do que outras possibilidades.

Com frequência, projetar um sistema que terá esforço de manutenção decrescente exige um esforço de implementação significativamente maior – exige-se muito mais trabalho e planejamento de projeto. Contudo, lembre-se de que o esforço de implementação é quase sempre um fator insignificante ao tomar decisões de projeto e deve geralmente ser ignorado.

Em suma:

É mais importante reduzir o esforço de manutenção do que reduzir o esforço de implementação.

Essa é uma das coisas mais importantes que há para saber sobre projeto de software.

Mas, o que causa o esforço de manutenção? Como projetamos sistemas cujo esforço de manutenção diminui com o tempo? Esse é o assunto de grande parte do restante deste livro. Mas, antes de chegarmos nisso, temos que examinar o futuro mais um pouquinho.

A qualidade do projeto

É muito fácil escrever software que ajuda uma pessoa, imediatamente. É muito mais difícil escrever software que ajuda milhões de pessoas agora e que continue a fazer isso durante décadas no futuro. Mas, onde vai estar a maioria do esforço de programação e quando a maioria desses usuários estará usando o software? Agora mesmo ou naquelas décadas por vir?

A resposta é que haverá muito trabalho de programação a ser feito – e muito mais usuários a ajudar – mais no futuro do que no presente. Seu software terá que competir e existir no futuro, e o esforço de manutenção e o número de usuários crescerão.

Quando ignoramos o fato de que há um futuro e fazemos as coisas que “apenas funcionam” no presente, nosso software fica difícil de se submeter à manutenção no futuro. Quando o software é difícil de se submeter à manutenção, é difícil fazê-lo continuar a ajudar as pessoas (uma de nossas metas em projeto de software). Se não puder adicionar novos recursos e não puder corrigir problemas, você acabará tendo um “software ruim”. Ele para de ajudar os seus usuários e fica cheio de erros.

Isso nos leva à seguinte regra:

O nível de qualidade de seu projeto deve ser proporcional à extensão do tempo futuro em que seu sistema continuará a ajudar as pessoas.

Se você estiver escrevendo software que será usado apenas nas próximas horas, não precisa colocar muito esforço no projeto. Mas, se seu software puder ser usado nos próximos dez anos (e isso acontece com muito mais frequência do que você poderia esperar, mesmo que você pense que só vai ser usado pelos próximos seis meses), então você tem que impor muito trabalho no projeto. Quando em dúvida, projete seu software como ele fosse ser usado durante muito, muito tempo: não se feche em nenhum método de fazer as coisas, mantenha-se flexível, não tome nenhuma decisão que não possa nunca mudar e dê muita atenção ao projeto.

Consequências imprevisíveis

Então, quando projetamos software, o futuro deve ser nosso foco principal. Contudo, uma das coisas mais importantes a saber sobre qualquer tipo de engenharia é essa:

Há algumas coisas sobre o futuro que você não sabe.

De fato, quando se trata de projeto de software, você simplesmente não pode saber a maioria das coisas sobre o futuro.

O erro mais comum e desastroso que os programadores cometem é prever algo sobre o futuro quando, na verdade, eles não podem saber.

Por exemplo, imagine que um programador tenha escrito um software em 1985 que corrigia disquetes com problemas. Ele não poderia corrigir nada mais – cada parte dele era totalmente dependente de como os disquetes funcionavam exatamente. Esse software agora seria obsoleto, porque as pessoas não usam mais disquetes. Aquele programador previu que “as pessoas sempre usarão disquetes” – algo que ele realmente não poderia saber.

Talvez seja possível prever o futuro de curto prazo, mas o futuro de longo prazo é grandemente desconhecido. O longo prazo também é mais importante para nós do que o curto prazo, porque nossas decisões de projeto terão mais consequências naquele período mais longo.



Você estará mais seguro se não tentar absolutamente prever o futuro e, em vez disso, tomar todas as decisões de projeto com base em informações conhecidas do presente.

Agora, isso pode soar como o exato oposto do que vimos dizendo até agora neste capítulo, mas não é. O futuro é a coisa mais importante a considerar ao tomar decisões de projeto. Mas há uma diferença entre projetar de modo a permitir alteração futura e tentar prever o futuro.

Como analogia, digamos que você tem uma escolha entre comer e morrer de fome. Você não precisa prever o futuro a fim de fazer essa escolha – você sabe que comer é a melhor escolha. Por quê? Porque ela o manterá vivo, e estar vivo contribui para um futuro melhor do que estar morto. O futuro é importante, e queremos considerá-lo em nossas decisões.

Escolhemos comer agora porque contribui para um futuro melhor. Mas o futuro não tem que ser previsto – não temos que dizer algo específico como: “Estou comendo agora porque amanhã terei que salvar a vida de um bebê”. Não importa o que aconteça amanhã, será um amanhã melhor se você comer agora em vez de morrer de fome.

De modo semelhante, no projeto de software podemos tomar certas decisões com base em informações que temos agora, para o propósito de fazer um futuro melhor (diminuindo o esforço de manutenção e aumentando o valor), sem ter que prever os detalhes do que vai acontecer nesse futuro.

Há limitadas exceções – às vezes, você sabe exatamente o que vai acontecer no futuro de curto prazo, e você pode tomar decisões com base nisso. Mas se for fazer isso, você deve estar muito certo sobre esse futuro, e ele deve estar ao alcance. Não importa quão inteligente você é, simplesmente não há um modo possível de prever com precisão futuros de longo prazo.

Vamos pegar um exemplo fora do ambiente da programação: CDs, que foram projetados em 1979 para substituir fitas cassete como o método principal de ouvir música. Quem poderia ter previsto que, 20 anos mais tarde, os DVDs seriam feitos do mesmo tamanho e formato de modo que os fabricantes pudessem fazer drives de CD/DVD para computadores? E quem poderia ter imaginado os problemas de girar um CD 50 vezes mais rápido do que se esperava que ele girasse quando era lido em um drive de CD-ROM?

É por isso que, em qualquer tipo de engenharia – incluindo a área de desenvolvimento de software – temos “princípios orientadores”. Esses são regras certas que, quando as seguimos, mantemos as coisas funcionando bem não importando o que aconteça no futuro. É isso que as leis e regras de projeto de software são – nossos “princípios orientadores” enquanto projetistas.

Então, sim, é importante lembrar que haverá um futuro. Mas isso não significa que você tem que prever esse futuro. Em vez disso, ele explica por que você deve tomar decisões de acordo com as leis e regras neste livro – porque elas levam a bom software futuro, não importa o que o futuro traga.

Não é nem mesmo possível prever todos os modos pelos quais uma lei ou regra particular poderá ajudar você no futuro – mas ela ajudará, e você ficará feliz por tê-la aplicado em seu trabalho.

Você é bem-vindo para discordar das leis, das regras e dos fatos que lê aqui. Por favor, chegue às suas próprias conclusões sobre eles. Mas você deve ser alertado de que, se não os seguir, provavelmente, vai acabar em uma grande confusão mais adiante, em um futuro que você não pode prever.

CAPÍTULO 5

Alteração

Agora que entendemos a importância do futuro e que há algumas coisas que não sabemos e não podemos saber sobre ele, o que podemos saber sobre ele?

Bem, uma coisa da qual você pode estar certo é que, com o passar do tempo, o ambiente em torno de seu software vai mudar. Nada permanece o mesmo para sempre. Isso quer dizer que seu software terá que mudar a fim de se adaptar ao ambiente em torno dele.

Isso nos dá a Lei da Alteração:

Quanto mais tempo seu programa existir, o mais provável é que qualquer parte dele tenha que ser alterada.

À medida que você entra em um futuro infinito, começa a tender para uma probabilidade de 100% de que cada parte de seu programa terá que mudar. Nos próximos cinco minutos, provavelmente nenhuma parte de seu programa terá que ser alterada. Nos próximos 10 dias, talvez uma pequena parte dele tenha que ser alterada. Nos próximos 20 anos, provavelmente a maior parte dele (se não todo ele) terá que ser alterado.

É difícil prever exatamente o que mudará e por quê. Talvez você tenha escrito um programa para carros de 4 rodas, mas no futuro todo mundo dirigirá caminhões de 18 rodas. Talvez você tenha escrito um programa para estudantes do ensino médio, mas o ensino médio ficará tão ruim que os estudantes não poderão mais entendê-lo.

A questão é: você não tem que tentar prever o que mudará; você apenas precisa saber que as coisas mudarão. Escreva seu software para que ele seja tão flexível quanto razoavelmente possível e será capaz de se adaptar a quaisquer futuras alterações que surjam.

Alteração em um programa do mundo real

Vamos olhar para alguns dados sobre como um programa do mundo real mudou com o tempo. Há centenas de arquivos neste programa particular, mas os detalhes para cada arquivo não caberão nesta página, então, quatro arquivos foram escolhidos como exemplos. Detalhes sobre esses arquivos são dados na tabela 5.1.

Tabela 5.1 – Alterações em arquivos no decorrer do tempo

Período analisado	Arquivo 1	Arquivo 2	Arquivo 3	Arquivo 4
Linhas originais	5 anos, 2 meses	8 anos, 3 meses	13 anos, 3 meses	13 anos, 4 meses
Linhas inalteradas	423	192	227	309
Linhas agora	271	101	4	8
Cresceram	664	948	388	414
Vezes alteradas	241	756	161	105
Linhas adicionadas	47	99	194	459
Linhas eliminadas	396	1,026	913	3,828
Linhas modificadas	155	270	752	3,723
Total de alterações	124	413	1,382	3,556
Razão de alteração	675	1,709	3,047	11,107
	1.6x	8.9x	13x	36x

Nessa tabela temos:

Período analisado

O período de tempo no decorrer do qual o arquivo existiu.

Linhas originais

Quantas linhas havia no arquivo quando ele foi originalmente escrito.

Linhas inalteradas

Quantas linhas são as mesmas agora em relação às linhas quando o arquivo foi originalmente escrito.

Linhas agora

Quantas linhas há no arquivo agora, no final do período de análise.

Cresceram

A diferença entre “Linhas agora” e “Linhas originais”.

Vezes alteradas

O número total de vezes que um programador fez um conjunto de alterações no arquivo (um conjunto de alterações envolve alterações em muitas linhas). Normalmente, um conjunto de alterações representará a correção de um erro, um novo recurso etc.

Linhas adicionadas

Quantas vezes, no histórico do arquivo, uma nova linha foi adicionada.

Linhas eliminadas

Quantas vezes, no histórico do arquivo, uma linha existente foi eliminada.

Linhas modificadas

Quantas vezes, no histórico do arquivo, uma linha existente foi alterada (mas não recém-adicionada ou eliminada).

Total de alterações

A soma das “Linhas adicionadas”, “Linhas eliminadas” e “Linhas modificadas” para cada arquivo.

Razão de alteração

O quanto “Total de alterações” é maior do que “Linhas originais”.

Quando nos referimos a “linhas” nas descrições, isso inclui cada linha nos arquivos: código, comentários, documentação e linhas em branco. Se você tivesse que fazer a análise sem considerar comentários, documentação e linhas em branco, uma diferença importante que você veria é que a contagem de “Linhas inalteradas” ficaria menor proporcionalmente aos outros números. (Em outras palavras, as “Linhas inalteradas” são quase sempre as de comentários, documentação ou linhas em branco).

O mais importante a perceber nessa tabela é que ocorrem muitas alterações em um projeto de software. Torna-se cada vez mais provável que qualquer linha de código particular mudará com o tempo, mas você não pode prever exatamente o que vai mudar, quando vai mudar ou quanto

terá que mudar. Cada um desses quatro arquivos mudou de modos muito diferentes (você pode ver isso mesmo apenas olhando para os números), mas todos eles mudaram uma quantidade significativa.

Há algumas outras coisas interessantes sobre os números, também:

- Olhando para a “Razão de alteração”, vemos que mais trabalho foi dedicado em alterar cada arquivo do que em escrevê-lo originalmente. Obviamente, contagens de linhas não são uma estimativa perfeita de quanto trabalho foi realmente realizado, mas elas nos dão, de fato, uma ideia geral. Às vezes, a razão é enorme – por exemplo, o arquivo 4 teve 36 vezes mais alterações totais do que linhas originais.
- O número de linhas inalteradas em cada arquivo é pequeno em comparação com sua contagem de “Linhas originais”, e até menor em comparação com sua contagem de “Linhas agora”.
- Muita alteração pode ocorrer em um arquivo mesmo se ele ficar apenas um pouco maior com o tempo. Por exemplo, o arquivo 3 cresceu apenas 161 linhas no decorrer de 13 anos, mas, durante esse tempo, a contagem de “Alterações totais” alcançou 3.047 linhas.
- A contagem de “Alterações totais” é sempre maior do que a contagem de “Linhas agora”. Em outras palavras, é mais provável você ter alterado uma linha em um arquivo do que ter uma linha em um arquivo, uma vez que o arquivo tenha estado por perto durante tempo suficiente.
- No arquivo 3, o número de “Linhas modificadas” é maior do que o número de linhas no arquivo original mais o número de “Linhas adicionadas”. As linhas desse arquivo foram modificadas com mais frequência do que novas linhas foram adicionadas. Em outras palavras, algumas linhas desse arquivo mudaram repetidas vezes. Isso é comum em projetos com vida longa.

Os pontos acima não representam tudo o que poderia ser aprendido aqui – há uma análise muito mais interessante que poderia ser feita sobre esses números. Você é levado a estudar mais esses dados (ou elaborar números semelhantes para seu próprio projeto) e ver o que mais você pode aprender.



Outra boa experiência de aprendizado é examinar o histórico de alterações feitas em um determinado arquivo. Se você tiver um registro de toda alteração feita em arquivos em seu programa e tiver um arquivo que tenha ficado disponível por muito tempo, tente olhar para cada alteração feita no decorrer da vida dele. Pense em se você poderia ter previsto essa alteração quando o arquivo foi originalmente escrito e considere se o arquivo poderia ter sido mais bem escrito originalmente para tornar as alterações mais simples. No geral, tente entender cada alteração e veja se você pode aprender qualquer coisa nova sobre desenvolvimento de software ao fazer isso.

As três falhas

Há três grandes erros que os projetistas de software cometem quando tentam confrontar a Lei da Alteração, listados aqui em ordem de quão comuns eles são:

1. Escrever código que não é necessário.
2. Não tornar o código fácil de alterar.
3. Ser genérico demais.

Escrever código que não é necessário

Há uma regra popular em projeto de software hoje chamada “Você Não Vai Precisar Disso”, ou YAGNI (na sigla em inglês), de forma abreviada. Essencialmente, essa regra declara que você não deve escrever código antes de realmente precisar dele. É uma boa regra, mas é nomeada de modo equivocado. Você realmente poderia precisar do código no futuro, mas, já que não pode prever o futuro, você ainda não sabe como o código precisaria trabalhar. Se o escrever agora, antes de precisar dele, você vai ter que reprojetá-lo para suas reais necessidades uma vez que você realmente começar a usá-lo. Então, poupe esse seu tempo de reprojeto, e simplesmente espere até precisar do código antes de escrevê-lo.

Outro risco de escrever código antes de precisar dele é que código não usado tende a desenvolver “bit rot”. Já que o código nunca é executado, ele pode aos poucos ficar fora de sintonia com o restante de seu sistema e assim desenvolver erros, e você nunca saberá. Então, quando começar a usá-lo, você terá que passar tempo depurando-o. Ou, pior ainda, talvez você confie no código nunca usado antes e não o verifica, e isso pode gerar erros para os usuários. De fato, a regra aqui deve realmente ser expandida para ser lida assim:

Não escreva código até realmente precisar dele, e elimine qualquer código que não esteja sendo usado.

Isto é, você também deve se livrar de qualquer código que não seja mais necessário. Você sempre poderá adicioná-lo de volta posteriormente se for necessário.

Há muitas razões pelas quais as pessoas acham que devem escrever código antes de ser necessário, ou manter por perto código que não esteja sendo usado. Em primeiro lugar, algumas pessoas acreditam que podem contornar a Lei da Alteração programando todo recurso que qualquer usuário possivelmente jamais poderia precisar, imediatamente. Então, elas pensam, o programa não terá que ser alterado ou melhorado no futuro. Mas isso está errado. Não é possível escrever um sistema que nunca mudará, desde que o sistema continue a ter usuários.

Outros acreditam que estão poupando tempo para si no futuro fazendo algum trabalho extra agora. Em alguns casos essa filosofia funciona, mas não quando você está escrevendo código que não é necessário. Mesmo que esse seja necessário no futuro, você quase certamente terá que passar tempo reprojetando-o, então, você está realmente perdendo tempo.

Escrevendo código desnecessário: um exemplo do mundo real

Certa vez um desenvolvedor – vamos chamá-lo de Max (ahã) – pensou equivocadamente que poderia ignorar essa regra. Em seu programa, havia boxes drop-down nas quais os usuários poderiam selecionar um valor. Toda empresa que usasse o programa poderia customizar a lista de opções exibida em cada box drop-down. Algumas empresas talvez quisessem que as opções fossem nomes de cores. Outras talvez quisessem que fossem nomes de cidades. Elas poderiam ser qualquer coisa. Então, a lista de opções válidas deveria ser armazenada em algum lugar em que cada empresa pudesse modificá-la.

O procedimento óbvio seria simplesmente armazenar a lista de valores e nada mais. Afinal de contas, isso era o necessário. Mas Max decidiu armazenar duas coisas: a lista de valores e também informações indicando se cada valor estava atualmente “ativo” – isto é, se os usuários poderiam atualmente selecionar esse valor ou se ele estava temporariamente desabilitado.

Contudo, Max nunca escreveu código para usar as informações que indicassem se cada campo estava ativo ou não. Todas as opções estavam ativas, o tempo todo, não importando o que os dados armazenados diziam. Ele estava certo de que estava prestes a escrever código para usar as informações “ativas” – talvez mesmo amanhã.

Vários anos se passaram e o código para manipular os dados “ativos” não foi escrito. Em vez disso, os dados simplesmente ficaram lá, sem uso, confundindo as pessoas e causando erros. Vários clientes e desenvolvedores escreveram a Max, perguntando por que nada aconteceu quando eles editaram manualmente a lista de valores e configuraram as opções como sendo inativas. Um desenvolvedor supôs erroneamente que o campo “ativo” estava em uso e escreveu um trecho de código que o usou, embora o restante do sistema não o tenha usado. Isso chegou aos clientes e eles começaram a relatar erros estranhos que exigiram muito trabalho para rastrear.

Posteriormente, algum desenvolvedor chegou e disse: “Hoje vou implementar a capacidade de desabilitar opções!” Contudo, ele descobriu que o campo “ativo” não estava projetado adequadamente conforme suas necessidades, então, ele teve que fazer muito trabalho de reprojeto para implementar seu recurso.

Resultado: muitos erros, muita confusão e trabalho extra para o desenvolvedor que acabou realmente precisando do código. E essa era uma violação relativamente menor da regra! Violações graves podem ter consequências consideravelmente piores, incluindo prazos perdidos, catástrofes e possivelmente até a destruição de seu projeto de software.

Não tornar o código fácil de alterar

Um dos grandes matadores de projetos de software é o que chamamos de “projeto rígido”. Isso ocorre quando um programador projeta código de um modo que é difícil alterar. Há dois modos de se criar um projeto rígido:

1. Fazer suposições demais sobre o futuro.
2. Escrever código sem projeto suficiente.

Exemplo: fazer suposições demais sobre o futuro

Uma agência do governo – vamos chamá-la de Hospital do Veterano – quer desenvolver um programa. Vamos chamar esse programa de “O Sistema de Cuidados com a Saúde”. Antes de fazer esse sistema, ela decide escrever um documento declarando como todo o sistema deve ser implementado. Ela passa um ano escrevendo esse documento, tomando toda decisão de forma individual sobre todo o sistema durante esse tempo.

Os desenvolvedores passam então três anos escrevendo o sistema de acordo com esse documento. Enquanto trabalham, eles descobrem que o projeto no documento é contraditório, incompleto e difícil de implementar. Mas o Hospital levou um ano inteiro para escrevê-lo – os desenvolvedores não podem esperar outro ano para ter que revisá-lo. Então eles implementam o sistema, seguindo o documento o mais próximo que podem.

O sistema é concluído e dado aos usuários pela primeira vez. Contudo, a situação no Hospital mudou dramaticamente nos últimos quatro anos, e quando os usuários começam a realmente usar O Sistema de Cuidados com a Saúde, eles percebem que querem algo completamente diferente. Mas o sistema é constituído de centenas de milhares de linhas de código, tudo projetado rigidamente de acordo com o documento – ele simplesmente não pode ser alterado sem meses ou anos de esforço.

Então o Hospital começa a escrever um novo documento para um novo sistema, e o processo recomeça novamente.

O erro do Hospital foi tentar prever o futuro. Eles supuseram que quaisquer decisões que tomassem no documento eram válidas para usuários reais e continuariam a ser válidas quando o sistema fosse concluído. Quando o verdadeiro futuro chegou, ele não era absolutamente como eles haviam previsto e seu sistema foi um fracasso de muitos milhões de dólares.

Uma solução melhor teria sido especificar apenas um recurso, ou um conjunto pequeno de recursos, e pedir imediatamente aos desenvolvedores para implementá-lo. Então, poderia ter havido um vaivém na comunicação e testes de usuário enquanto ocorria o desenvolvimento. Quando o primeiro conjunto de recursos foi concluído e liberado, eles poderiam ter trabalhado em recursos adicionais, um de cada vez, até que posteriormente tivessem um sistema que fosse bem projetado e satisfizesse plenamente as necessidades de seus usuários.

Exemplo: código sem projeto suficiente

Pede-se a um desenvolvedor para criar um programa que as pessoas possam usar para acompanhar tarefas que elas precisam executar. Para criar uma nova “tarefa” no sistema, os usuários preenchem um formulário com algumas informações, como um resumo da tarefa e quanto adiantados eles estão nela. Isso armazena dados em um banco de dados. Então, eles podem fazer anotações sobre seu progresso na tarefa com o passar do tempo e, posteriormente, anotar que concluíram a tarefa.

Há um campo chamado “Status” que indica quanto adiantado o usuário está na execução da tarefa. Os valores para esse campo são “Tarefa não executada”, “Em andamento”, “Suspensa” e “Concluída”. Quando o campo Status tem o valor “Tarefa não executada”, ele pode mudar apenas para “Em andamento”. Quando o campo Status tem o valor “Em andamento”, ele pode mudar para “Suspensa” ou “Concluída”. E quando tem o valor “Concluída”, ele pode mudar apenas de volta para “Em andamento”.

Há 10 outros campos neste programa com regras semelhantes. Cada um deles contém alguma informação diferente sobre a tarefa (por exemplo, a quem ela é designada, qual é o prazo etc.).

Para implementar essas regras, o desenvolvedor escreve um trecho de código muito longo e contínuo sem nenhuma estrutura, em um único arquivo. Ele valida cada campo com código customizado que é específico daquele campo. Por exemplo, toda vez que ele precisa verificar se o status é “Concluída”, escreve literalmente a palavra “Concluída” no código. Além disso, o código não é escrito para ser reutilizável. Onde o programa tem campos semelhantes, o desenvolvedor corta e cola código e então o modifica um pouco para o novo campo.

O código funciona. O arquivo tem 3 mil linhas. Ele necessita quase inteiramente de um projeto.

Vários meses mais tarde, esse desenvolvedor deixa o projeto.

Um novo desenvolvedor chega e é designado para fazer a manutenção desse projeto. Ele rapidamente descobre que esse código é difícil de alterar – se ele altera uma parte do código, também tem que alterar muitas outras partes do mesmo modo a fim de mantê-lo funcionando. Para piorar ainda mais as coisas, as diversas partes estão espalhadas sem nenhuma explicação ou sistema lógico – você simplesmente precisa ler o arquivo inteiro toda vez para fazer uma alteração.

Os clientes começam a pedir novos recursos. A princípio, o novo desenvolvedor faz o possível para implementar esses novos recursos. Ele adiciona ainda mais código a esse arquivo. Ele acaba tendo 5 mil linhas.

Posteriormente, os clientes começam a pedir recursos que simplesmente não podem ser implementados com esse projeto. Eles querem enviar informações sobre as tarefas por email, mas esse código apenas trabalha com um formulário. Ele é todo projetado muito especificamente em torno de como o formulário funciona – ele jamais trabalharia com email.

Começam a aparecer concorrentes que podem atualizar tarefas por email. O projeto começa a perder seus clientes.

A única razão pela qual esse projeto sobrevive é que dois desenvolvedores passam um ano inteiro reprojetando apenas esse arquivo para que ele possa ser facilmente alterado. Eles fazem o possível para atender a outras solicitações de recursos enquanto estão reprojetando, mas a maior parte do tempo é gasto em reprojetar.

A regra usada para evitar projeto rígido é:

Código deve ser projetado com base no que você sabe agora, não no que você pensa que acontecerá no futuro.

Projete apenas com base em seus requisitos imediatos e conhecidos, sem descartar a possibilidade de requisitos futuros. Se você sabe com certeza que precisa que o sistema faça X, e apenas X, então projete-o apenas para fazer X, agora mesmo. Ele talvez faça outras coisas que não sejam X no futuro, e você deve ter isso em mente, mas, por enquanto, o sistema deve apenas fazer X.

Enquanto estiver projetando dessa forma, isso o ajudará a manter pequenas as suas alterações individuais. Quando você apenas tem que fazer uma pequena alteração, é fácil fazer algum verdadeiro projeto nela.

Isso não significa que planejamento é ruim. Certa quantidade de planejamento é muito valiosa em projeto de software. Mas mesmo que você não elabore planos detalhados, estará bem desde que suas alterações sejam sempre pequenas e que seu código permaneça facilmente adaptável para o futuro desconhecido.

Sendo genérico demais

Quando confrontados com o fato de que seu código mudará no futuro, alguns desenvolvedores tentam resolver o problema projetando uma solução tão genérica que (eles acreditam) acomodará toda possível situação futura. Chamamos isso de “superengenharia”.

O dicionário define superengenharia como uma combinação de “super” (significando “demais”) e “engenharia” (significando “projeto e construção”). Então, segundo o dicionário, o termo significa projetar ou construir demais para sua situação.

Espere – projetar ou construir demais? O que é “demais”? Projetar não é uma coisa boa?

Bem, sim, a maioria dos projetos poderia investir mais tempo da fase de “projeto”, como vimos no “Exemplo: Código sem projeto suficiente” na página 58. Mas, de vez em quando, alguém fica realmente envolvido e acaba extrapolando – meio que construindo um laser orbital para

destruir um formigueiro. Um laser orbital é uma incrível façanha de engenharia, mas custa uma enorme quantia de dinheiro, leva muito tempo para ser construído e é um pesadelo em termos de manutenção. Você pode imaginar ter que subir até lá e consertá-lo quando quebrar?

Há vários outros problemas com a superengenharia:

1. Você não pode prever o futuro, então não importa quão genérica é sua solução, ela não será genérica o bastante para satisfazer os verdadeiros requisitos futuros que você terá.
2. Quando seu código é genérico demais, ele frequentemente não manipula detalhes muito bem sob o ponto de vista do usuário. Por exemplo, digamos que você projeta algum código que trata toda entrada do mesmo modo – tudo não passa de bytes. Às vezes esse código processa texto, e às vezes processa imagens, mas tudo que ele sabe é que está recebendo bytes. De certa forma, esse é um bom projeto: o código é simples, independente, pequeno etc.

Mas então você se certifica de que nenhuma parte de seu código distingue entre imagens e texto. Isso é genérico demais. Quando o usuário introduz uma imagem ruim, o erro que ele obtém é: “Você introduziu bytes ruins”. Deveria ter dito: “Você introduziu uma imagem ruim”, mas seu código é tão genérico que não pode dizer isso ao usuário. (Há muitos modos pelos quais esse código genérico pode ser insuficiente quando submetido a usos específicos; esse é apenas um exemplo).

3. Ser genérico demais envolve escrever muito código que não é necessário, o que nos leva à nossa primeira falha.

Em geral, quando seu projeto torna as coisas mais complexas em vez simplificá-las, você está fazendo superengenharia. Aquele laser orbital complicaria enormemente a vida de uma pessoa que precisasse apenas destruir alguns formigueiros, ao passo que um simples veneno contra formigas simplificaria grandemente a vida dessa pessoa em eliminar o problema das formigas (supondo que funcionasse).

Ser genérico com as coisas certas, das maneiras certas, pode ser a base de um projeto de software bem-sucedido. Contudo, ser genérico demais pode ser a causa de complexidade inédita, confusão e esforço

de manutenção. A regra para evitar essa falha é semelhante à regra para evitar projetos rígidos:

Seja apenas tão genérico quanto você sabe que precisa ser, imediatamente.

Exemplo: sendo genérico demais

Em uma parte de certo programa, o usuário preencheu um formulário e o programa enviou centenas de e-mails. Essa parte do programa era muito lenta. O usuário submetia o formulário e o programa ficava lá durante muito tempo, enviando todas as mensagens.

Para tornar isso mais rápido, os desenvolvedores decidiram não enviar todos os emails imediatamente. Em vez disso, eles seriam enviados em background depois de o usuário ter submetido o formulário, usando um trecho de código preexistente chamado “Email Sender”.

O desenvolvedor que começou a trabalhar nessa alteração decidiu que algumas empresas talvez quisessem usar algo além do Email Sender. Ele escreveu centenas de linhas de código para permitir que os clientes “se conectassem” a outros sistemas para fazer trabalho em background. Nenhum cliente havia pedido isso; o desenvolvedor simplesmente previu que alguém iria querer esse tipo específico de flexibilidade no futuro.

Posteriormente, o arquiteto-chefe do programa assumiu o trabalho dessa alteração. Ele eliminou todo o código para “se conectar” a outros sistemas, porque não havia evidência de que os usuários o quisessem. Assim, não havia evidência de que o código deveria ser tão genérico imediatamente. Com esses trechos eliminados, a alteração se tornou muito mais simples.

Quatro anos se passaram desde que a alteração foi originalmente feita, e nem um único cliente precisou da capacidade de se conectar a outros sistemas. Não houve, na verdade, razão para ser tão genérico.

Desenvolvimento e projeto incrementais

Há um método de desenvolvimento de software que evita as três falhas por sua exata natureza, chamado “desenvolvimento e projeto incrementais”. Ele envolve projeto e desenvolvimento de um sistema parte por parte, em ordem.

É mais fácil explicar com um exemplo. Aqui está como o usariamos para desenvolver um programa de calculadora que precisa adicionar, subtrair, multiplicar e dividir:

1. Planeje um sistema que faz apenas adição e nada mais.
2. Implemente esse sistema.
3. Organize o projeto do sistema para que seja fácil adicionar um recurso de subtração.
4. Implemente o recurso de subtração no sistema. Agora temos um sistema que faz apenas adição e subtração e nada mais.
5. Organize o projeto do sistema novamente para que seja fácil adicionar um recurso de multiplicação.
6. Implemente o recurso de multiplicação no sistema. Agora temos um sistema que faz adição, subtração, multiplicação e nada mais.
7. Organize o projeto do sistema novamente para que seja fácil adicionar o recurso de divisão. (Nessa altura, isso deve exigir pouco ou nenhum esforço, porque já melhoramos o projeto antes de implementar a subtração e a multiplicação).
8. Implemente o recurso de divisão no sistema. Agora temos o sistema que pretendíamos desenvolver, com um excelente projeto que lhe serve bem.

Esse método de desenvolvimento exige menos tempo e menos pensamento do que planejar todo o sistema antecipadamente e desenvolvê-lo de uma vez. Pode não ser fácil a princípio se você estiver acostumado a outros métodos de desenvolvimento, mas ele se tornará fácil com a prática.

A parte complicada de usar este método é decidir sobre a ordem da implementação. Em geral, você deve selecionar o que for mais simples de se trabalhar em cada etapa, quando você chegar lá. Selecionamos adição primeiro porque era a mais simples das quatro operações, subtração em segundo lugar porque ela se fundamenta logicamente na adição de um modo muito simples. Poderíamos possivelmente ter selecionado multiplicação em segundo lugar, já que a multiplicação é apenas a ação de fazer adição muitas vezes. A única coisa que não teríamos selecionado

em segundo lugar é a divisão, porque passar da adição para a divisão é distante demais de um passo lógico – é complexo demais. Por outro lado, passar da multiplicação para a divisão no final realmente era muito simples, então, essa era uma boa escolha.

Às vezes você pode até precisar selecionar um recurso individual e dividi-lo em etapas pequenas, simples e lógicas para que ele possa ser implementado.

Isso é, na verdade, uma combinação de dois métodos: um chamado “desenvolvimento incremental” e outro chamado “projeto incremental”. Desenvolvimento incremental é um método de criar um sistema fazendo o trabalho em pequenas partes. Em nossa lista, cada etapa que começou com “Implemente” era parte do processo de desenvolvimento incremental. Projeto incremental é similarmente um método de criar e melhorar o projeto do sistema em pequenos incrementos. Cada etapa que começou com “Organize o projeto do sistema” ou o “Plano” era parte do processo de projeto incremental.

Desenvolvimento e projeto incrementais não é o único método válido de desenvolvimento de software, mas é um que definitivamente previne as três falhas apresentadas na seção anterior.

CAPÍTULO 6

Erros e projeto

Infelizmente, nenhum programador é perfeito. Bons programadores introduzirão aproximadamente um erro em um programa para cada 100 linhas de código que escrevem. Os melhores programadores, sob as melhores circunstâncias, introduzirão um erro para cada mil linhas de código que escrevem.

Em outras palavras, não importa quanto bom ou ruim você é como programador, é certo que quanto mais você codificar, mais erros introduzirá.

Isso nos permite declarar uma lei chamada Lei da Probabilidade de Erros:

A chance de introduzir um erro em seu programa é proporcional à quantidade de alterações que você faz nele.

Isso é importante porque os erros violam nosso propósito de ajudar as pessoas e, portanto, devem ser evitados. Além disso, corrigir erros é uma forma de manutenção. Assim, aumentando o número de erros, aumenta nosso esforço de manutenção.

Com essa lei, sem ter de prever o futuro, podemos imediatamente perceber que fazer pequenas alterações tem a probabilidade de levar a menor esforço de manutenção do que fazer grandes alterações. Pequenas alterações = menos erros = menos manutenção.

Essa lei também é às vezes declarada mais informalmente como “Você não pode introduzir novos erros se não adicionar ou modificar código”.

O curioso sobre essa lei é que ela parece estar em conflito com a Lei da Alteração – seu software tem que mudar, mas alterá-lo introduzirá erros. Esse é um conflito verdadeiro, e o equilíbrio dessas leis é o que exige sua inteligência como projetista de software. É realmente esse conflito

que explica por que precisamos de projeto e, de fato, nos diz qual é o projeto ideal é:

O melhor projeto é o que permite a máxima alteração no ambiente com a mínima alteração no software.

E isso, simplesmente, resume muito do que é conhecido sobre bom projeto de software hoje.

Se não estiver com erro...

Ok, então você não pode introduzir erros em seu programa se você não adicionar ou modificar o código, e essa é uma lei importante do projeto de software. Contudo, há também uma regra muito importante relacionada que muitos engenheiros de software ouviram de uma forma ou de outra, mas às vezes esquecem:

Nunca “corija” nada a menos que seja um problema e você tenha evidência mostrando que o problema realmente existe.

É importante ter evidência dos problemas antes de tratar deles. Caso contrário, talvez você esteja desenvolvendo recursos que não resolvem o problema de ninguém, ou talvez você esteja “corrigindo” coisas que não estão com erro.

Se você corrigir problemas sem evidência, provavelmente vai danificar coisas. Você está introduzindo alteração em seu sistema, o que vai trazer novos erros junto. E não apenas isso, mas você está perdendo tempo e adicionando complexidade a seu programa sem nenhum motivo.

Então, o que conta como “evidência”? Suponha que cinco usuários relatam que, quando eles pressionam o botão vermelho, seu programa trava. Ok, isso é evidência suficiente! Alternativamente, você mesmo pode pressionar o botão vermelho e observar que o programa trava.

Contudo, só porque um usuário relata algo, não significa que seja um problema. Às vezes, o usuário simplesmente não percebeu que o programa já tinha algum recurso, e então pediu para você implementar algo mais sem necessidade. Por exemplo, digamos que você escreva um

programa que ordena uma lista de palavras alfabeticamente, e um usuário peça que você adicione um recurso que ordene uma lista de letras alfabeticamente. Seu programa já faz isso. Na verdade, ele já faz mais do que isso – esse frequentemente é o caso, com esse tipo de solicitação confusa. Nesse caso, o usuário pode pensar que há um problema quando não há. Ele pode até apresentar a “evidência” de que não pode ordenar uma lista de letras, quando, de fato, o problema é apenas que ele não percebeu que deveria usar o recurso de ordenação de palavras.



Se você recebe muitas solicitações como as anteriores, isso significa que os usuários não podem encontrar facilmente os recursos de que precisam em seu programa. Isso é algo que você deve corrigir.

Às vezes, um usuário relatará que há um erro, quando na verdade é o programa comportando-se exatamente como você pretendia que ele o fizesse. Nesse caso, é uma questão de regras da maioria. Se um número significativo de usuários pensa que o comportamento é um erro, então é um erro. Se apenas uma minoria (como um ou dois) pensa que é um erro, então não é um erro.

O erro mais famoso nessa área é o que chamamos de “otimização prematura”. Isto é, alguns desenvolvedores parecem gostar de fazer as coisas rápido, mas eles passam tempo otimizando seu código antes de saberem que ele está lento! É como uma instituição benéfica enviando comida a pessoas ricas e dizendo: “Só queríamos ajudar as pessoas!”. Ilógico, não é? Eles estão resolvendo um problema que não existe.

As únicas partes de seu programa com as quais você deve se preocupar com velocidade são as partes exatas que você pode mostrar que estão causando um verdadeiro problema de desempenho para seus usuários. Para o restante do código, as principais preocupações são flexibilidade e simplicidade, não fazê-lo ir rápido.

Há muitos modos de violar essa regra, mas o jeito de segui-la é simples: apenas obtenha a verdadeira evidência de que um problema é válido antes de tratar dele.

Não se repita

Essa é provavelmente a mais conhecida regra em projeto de software. Esse livro não é o primeiro lugar em que ela já apareceu. Mas é válida, e por isso é incluída aqui:

Em qualquer sistema particular, qualquer informação deve, idealmente, existir apenas uma vez.

Digamos que você tem um campo chamado “Password” que aparece em 100 telas da interface do usuário de seu programa. E se você quiser alterar o nome do campo para “Passcode”? Bem, se você tiver armazenado o nome do campo em um local central em seu código, corrigi-lo exigirá a alteração de uma linha apenas. Mas se você escreveu a palavra “Password” manualmente em todas as 100 telas da interface do usuário, precisará fazer 100 alterações para corrigi-la.

Isso também se aplica a blocos de código. Você não deve copiar e colar blocos de código. Em vez disso, você deve usar os diversos recursos da programação que permitem a um trecho de código ser “usado”, “chamado” ou “incluído” em outro trecho de código existente.

Uma das boas razões para seguir essa regra é a Lei da Probabilidade de Erros. Se podemos reutilizar código antigo, não temos que escrever ou alterar tanto código quando adicionamos novos recursos, então introduzimos menos erros.

Ela também nos ajuda com a flexibilidade em nossos projetos. Se precisamos alterar o modo como nosso programa funciona, podemos alterar algum código em apenas um lugar, em vez percorrer todo o programa e fazer diversas alterações.

Projetos muito bons baseiam-se nessa regra. Isto é, quanto mais atento você puder ficar em fazer um código ser “usado” por outro e centralizar informações, melhor será seu projeto. Essa é outra área onde sua inteligência realmente desempenha um papel na programação.

CAPÍTULO 7

Simplicidade

Ok, então, se nunca alterarmos nosso software, poderemos evitar totalmente os erros. Mas a alteração é inevitável, particularmente se vamos adicionar novos recursos. Então, “não altere nada” não pode ser a técnica definitiva de redução de erros.

Conforme explicado no capítulo 6, se quer evitar erros em seu código, realmente ajuda manter suas alterações pequenas. Mas, se quer ir além e eliminar erros mesmo de suas pequenas alterações, há outra lei que pode ajudá-lo. E ela não reduz simplesmente os erros – ela mantém seu código sujeito à manutenção, facilita a inclusão de novos recursos e melhora a compreensão geral de seu código. Essa é a Lei da Simplicidade:

A facilidade de manutenção de qualquer software é proporcional à simplicidade de suas partes individuais.

Isto é, quanto mais simples são as partes, mais facilmente você poderá alterar as coisas no futuro. Perfeita facilidade de manutenção é impossível, mas é a meta que você pretende atingir – alteração total ou código novo infinito sem nenhuma dificuldade.

Talvez tenha notado que essa lei não fala da simplicidade de todo o sistema, contudo, apenas das partes individuais. Por quê?

Bem, um programa de computador de tamanho médio é tão complexo que nenhuma pessoa poderia comprehendê-lo todo de uma vez. É apenas possível compreender partes dele. Então, realmente sempre temos alguma estrutura grande e complexa para todo o nosso programa. O que então se torna importante é que as partes podem ser entendidas quando olhamos para elas. Quanto mais simples são as partes, mais

provável é que qualquer pessoa as entenderá. Isso é particularmente importante quando você está passando seu código para outras pessoas, ou quando se afasta de seu código por alguns meses e então tem que voltar e repreender o que você fez.

Uma analogia com a arquitetura

Imagine que você está construindo uma estrutura de aço de 9 metros de altura. Poderia fazê-la com várias vigas pequenas, que são partes simples. Ou poderia forjar três enormes e complexas partes de aço e uni-las.

Com a abordagem das vigas, é fácil fazer ou comprar as partes individuais. E, se uma quebrar, você simplesmente a substitui por uma peça de reposição idêntica. A construção é simples, assim como a manutenção.

As três partes enormes, por outro lado, devem ser cuidadosamente feitas sob medida e trabalhadas extensamente. Cada parte concluída é tão grande que é difícil encontrar e consertar todos os seus defeitos. E se, depois que a construção estiver terminada, você descobrir numerosas falhas em cada parte, você não poderá substituí-las – a construção desabararia se você tirasse qualquer parte. Então você tem que soldar com feios remendos de metal e esperar que a coisa toda permaneça de pé.

Software é muito semelhante – quando você escreve seu código em partes simples e independentes, corrigir erros e fazer a manutenção do sistema é fácil. Quando você projeta trechos grandes e complexos, cada parte exige muito trabalho e não recebe tanto acabamento quanto deveria. O sistema se torna difícil de se submeter à manutenção, e patches têm que ser adicionados constantemente para mantê-lo funcionando.

Então, por que as pessoas às vezes escrevem trechos grandes e complexos em vez de partes pequenas e simples? Bem, há uma economia de tempo percebida com o método das partes enormes quando você está criando o software pela primeira vez. Com várias partes pequenas, muito tempo é gasto unindo-as. Você não vê isso com as partes enormes – há algumas delas, elas se unem de repente, e é isso.

Contudo, a qualidade do sistema de partes enormes é muito mais baixa, e você passará muito tempo corrigindo-o no futuro. Ficará cada vez mais difícil fazer a manutenção, enquanto o sistema simples se torna cada vez mais fácil. No longo prazo, é a simplicidade que é eficiente, não a complexidade.

Então, como usamos essa lei, no mundo prático da programação? Esse é o assunto a ser tratado no restante deste livro. Em geral, contudo, a ideia é tornar os componentes individuais de seu código os mais simples possíveis e então certificar-se de que eles permaneçam assim com o tempo.

Um bom modo de fazer isso é usar o método do desenvolvimento e projeto incrementais introduzido no fim do capítulo 5. Já que há uma etapa de “reprojeto” antes que cada novo recurso seja adicionado, você pode usar esse tempo para simplificar o sistema. Mesmo que não esteja usando esse método, contudo, você pode levar algum tempo entre adicionar recursos para simplificar quaisquer partes que parecem complexas demais para você ou seus colegas desenvolvedores.

De uma forma ou de outra, você frequentemente tem que pegar o que criou e torná-lo mais simples – você não pode partir do princípio que o seu projeto inicial seja sempre o certo. Você tem que reprojetar partes do sistema continuamente à medida que surgem novas situações e requisitos.

Admito, essa pode ser uma tarefa razoavelmente difícil. Você nem sempre recebe ferramentas simples para escrever seus programas – as linguagens são complexas, o próprio computador é complexo etc. Mas esforce-se para obter simplicidade com o que você tem.

Simplicidade e a equação do projeto de software

Talvez você tenha percebido isso, mas essa lei nos diz a coisa mais importante que podemos fazer imediatamente que reduzirá o esforço de manutenção na Equação do Projeto de Software – tornar nosso código mais simples. Não temos que prever o futuro para fazer isso; podemos apenas olhar para o nosso código, ver se ele é complexo e torná-lo menos complexo para nós mesmos imediatamente. É assim que você obtém um esforço de manutenção que diminui com o tempo – você trabalha continuamente para tornar seu código mais simples.

Há certa quantidade de trabalho envolvido ao fazer essa simplificação, mas, no geral, é muito mais fácil fazer alterações em um sistema simples do que em um sistema complexo – então, você passa um pouco de tempo fazendo a simplificação agora para poupar muito tempo mais tarde.

À medida que você diminui o esforço de manutenção para seu sistema, você aumenta a desejabilidade por todas as alterações possíveis. (Volte ao capítulo 4 e dê outra olhada na Equação do Projeto de Software se quiser refrescar sua memória sobre os detalhes). Simplificar seu código diminui o esforço de manutenção, aumentando assim a desejabilidade por toda outra possível alteração.

A simplicidade é relativa

Ok, então, queremos que as coisas sejam simples. Contudo, como você define “simples” realmente depende de seu público-alvo. O que é simples para você poderia não ser para seus colegas de trabalho. Além disso, quando você cria algo, pode parecer relativamente “simples” para você, porque você o entende por dentro e por fora. Mas, para alguém que nunca o viu antes, poderia parecer muito complicado.

Se você quiser entender o ponto de vista de alguém que não sabe nada sobre seu código, encontre algum código que você nunca leu e leia-o. Tente entender não só as linhas individuais, mas o que todo o programa está fazendo e como você o modificaria se tivesse que fazê-lo. Essa é a mesma experiência que outras pessoas têm quando leem seu código. Talvez você note que o nível de complexidade não tem que ser muito alto antes de se tornar frustrante ler o código de outras pessoas.

É por isso que é bom ter seções na documentação de seu código como “Novo Nesse Código?” que contenham algumas explicações simples que ajudarão as pessoas a entender seu código. Essas devem ser escritas como se o leitor nada soubesse sobre o programa, porque se as pessoas são novas em algo, elas provavelmente nada sabem a respeito.

Muitos projetos de software confundem isso. Você vai ler a documentação escrita para os desenvolvedores e se depara com uma quantidade enorme de links e nenhuma direção. Isso parece simples para o desenvolvedor experiente no projeto, porque uma página com muitos links permite que o desenvolvedor vá rapidamente para a parte que ele está procurando. Mas, para alguém novo no projeto, não é muito fácil. Por outro lado, para

o desenvolvedor experiente, adicionar uma página com botões grandes e simples e eliminar essa lista de links contribuiriam para a complexidade de sua tarefa, porque sua meta principal será apenas encontrar algo muito específico bem rápido na documentação.

A única coisa pior do que documentação complexa é nenhuma documentação, onde apenas se espera que você entenda para você mesmo ou “já saiba” como o código funciona. Para o desenvolvedor, o modo pelo qual seu programa funciona é óbvio, mas para os outros é totalmente desconhecido.

O contexto é importante, também. Por exemplo, no contexto do código do programa, tecnologias avançadas frequentemente levam à simplicidade, se usadas adequadamente. Mas, imagine se a estrutura interna avançada de tal programa fosse exibida diretamente em uma web page como a única interface do programa – ela não seria simples naquele contexto, mesmo para o desenvolvedor!

Às vezes, o que parece complexo em um contexto é simples em outro. Exibir muito texto explicativo em um outdoor ao lado da estrada seria demasiadamente complexo – simplesmente não há tempo para que os motoristas que passam leiam todo aquele texto, então, seria estúpido colocá-lo lá. Mas, em um manual para um programa de computador, incluir muito texto explicativo seria muito mais simples do que apenas dar uma descrição de algo em uma frase. É por isso que este livro não tem capítulos de apenas uma linha; realmente não seria tão simples apenas dizer algo e então não explicá-lo.

Com todos esses pontos de vista diferentes e contextos a considerar, isso significa que alcançar simplicidade é tão difícil? Não! Absolutamente. Há públicos-alvo específicos para tudo, e o contexto de qualquer coisa individual que você está fazendo normalmente é muito limitado. O problema é sempre solucionável. É importante levar essas considerações em conta quando projetar seu software, para que, quando alguém for usá-lo, ele seja realmente simples para aquela pessoa particular.

A guerra dos editores

Tem havido numerosas discussões no mundo do desenvolvimento de software sobre quais são as melhores ferramentas para um trabalho. As pessoas adoram diferentes editores de texto, diferentes linguagens de programação, diferentes sistemas operacionais etc. Talvez a “guerra” mais famosa em desenvolvimento de software seja entre os usuários de dois editores de texto particulares, vi e Emacs. Os usuários de cada um têm alegado às vezes que seu editor preferido é fundamentalmente superior ao outro.

Na realidade, raramente há uma ferramenta fundamentalmente superior para escrever software; há apenas uma ferramenta que as pessoas em particular acham mais simples para a tarefa à mão. Usuários de Emacs acham o Emacs a ferramenta mais simples de usar para escrever software, e usuários de vi acham o vi a mais simples. Até certo ponto, isso tem a ver com diferenças fundamentais entre pessoas em termos de como elas gostam de trabalhar ou como elas pensam. As pessoas simplesmente têm preferências diferentes e não existe certo ou errado. Mas, em um grau maior, a simplicidade percebida de uma ferramenta tem a ver com familiaridade – qualquer pessoa que tenha usado uma ferramenta em particular durante muito tempo provavelmente ficou muito familiarizado com ela, o que a torna muito mais simples do que qualquer outra ferramenta, do ponto de vista daquela pessoa. A fim de que uma nova ferramenta pareça igualmente simples, essa ferramenta teria que ser extremamente simples, e os editores de texto dos programadores raramente o são.

Não programadores provavelmente considerariam ambos os editores de texto irracionalmente complexos, o que é outro exemplo de como a simplicidade é relativa.

Ferramentas podem ter problemas que as tornam inadequadas para a tarefa à mão ou a escolha errada por razões de projeto de software (ver “Tecnologias ruins” na página 89 no capítulo 8). Mas, excluindo esses problemas, a simplicidade relativa de uma ferramenta é o que permitirá a um programador determinar o que é melhor para uma dada situação.



Quão simples você tem que ser?

Quando você está trabalhando em um projeto, podem surgir perguntas sobre simplicidade. Quão simples nós realmente temos que ser? Apenas quanto temos que simplificar isso? É simples o bastante?

Bem, claro, a simplicidade é relativa. Mas, mesmo assim, você ainda pode alcançar mais ou menos simplicidade. Do ponto de vista relativo de seu usuário, seu produto poder ser difícil de usar, fácil de usar ou algo entre os dois. Igualmente, do ponto de vista de outro programador, seu código pode ser relativamente difícil ou fácil de ler.

Então, quão simples você tem que ser?

Honestamente?

Se você realmente quer ter sucesso?

Muito, muito simples.

A boa coisa nesse nível de simplicidade é que, na maioria dos casos, qualquer coisa usável por pessoas normais também é usável por gênios. Você abrange uma gama muito mais ampla de possíveis usuários.

Mas, com frequência, na verdade, as pessoas simplesmente não entendem quão muito simples elas têm que ser para chegar àquele nível. Vejamos um exemplo. Quando você está no shopping, há mapas que lhe dizem onde tudo está. Nos melhores mapas de shopping, há um enorme ponto vermelho, com a frase “VOCÊ ESTÁ AQUI” em letras gigantescas, bem na sua frente. Nos mapas mais simples, há um minúsculo triângulo amarelo no meio do mapa que é muito difícil de encontrar, e mais para o lado há algum texto que explica: “O minúsculo triângulo amarelo significa ‘Você está aqui!’”. Junte isso à confusão geral de tentar encontrar qualquer coisa nesses mapas e você poderia passar cinco ou seis minutos simplesmente em pé em frente da coisa, tentando entender como chegar aonde você vai.

Para quem projetou o mapa, isso pode parecer totalmente razoável. Ele passou muito tempo projetando-o, então era claramente importante para que ele ficasse feliz em passar vários minutos olhando para o mapa, aprendendo tudo sobre ele, entendendo-o etc. Mas, para nós, as

pessoas que estão realmente usando o mapa, essa é uma parte muito insignificante de nossa existência. Queremos apenas que ele seja o mais simples possível, para que possamos usá-lo rapidamente e continuar com nossas vidas!

Muitos programadores são particularmente maus em relação a isso com seu código. Eles supõem que outros programadores estarão dispostos a passar muito tempo aprendendo tudo sobre seu código, porque, afinal de contas, levou muito tempo escrevê-lo! O código é importante para eles, então, não será importante para todo mundo?

Agora, programadores são em geral um grupo inteligente. Mas ainda é um erro pensar: "Oh, outros programadores vão entender tudo que fiz aqui sem nenhuma simplificação ou explicação do meu código". Não é uma questão de inteligência – é uma questão de conhecimento. Programadores que são novos ao seu código nada sabem sobre ele; eles têm que aprender. Quanto mais fácil você tornar para eles aprenderem, mais rápido eles vão entendê-lo e mais fácil será para eles o usarem.

Há muitos modos de tornar seu código fácil de aprender: documentação simples, projeto simples, tutoriais passo a passo etc.

Mas, se seu código não for extremamente simples de aprender, as pessoas vão ter problemas com ele. Elas vão usá-lo incorretamente, criar erros e geralmente bagunçar as coisas. E, quando tudo isso acontece, a quem elas vão perguntar a respeito? Sim, a você! Você vai passar tempo respondendo todas as perguntas delas. (Mmm, parece divertido, não?)

Nenhum de nós gosta de ser menosprezado ou tratado como idiota. Às vezes, isso nos leva a criar coisas que são um pouco complicadas, para que tenhamos a sensação de que não estamos menosprezando o usuário ou outros programadores. Incluímos algumas palavras difíceis, fazemos as coisas um pouco menos simples e as pessoas respeitam nossa inteligência, mas sentem-se meio estúpidas porque não entendem. Elas poderiam pensar que somos muito mais inteligentes do que elas jamais poderiam ser e isso é meio lisonjeiro. Mas, de fato, isso as está ajudando?

Por outro lado, quando você torna seu produto ou código extremamente simples, você está permitindo que as pessoas o entendam. Isso as faz sentir-se inteligentes, permite que elas façam o que estão tentando fazer

e absolutamente não reflete de forma negativa sobre você. Na verdade, as pessoas provavelmente o admirarão mais se você tornar as coisas simples do que se você as tornar complexas.

Agora, toda a sua família não tem que ser capaz de ler seu código. A simplicidade ainda é relativa, e o público-alvo para código são outros programadores. Mas, para esses outros programadores, seu código deve parecer muito simples e fácil de entender. Ele pode usar tanta tecnologia avançada quanto seja exigida para alcançar essa simplicidade, mas ele ainda deve ser definitivamente simples.

Quando surgir a pergunta: "Quão simples tenho que ser?", você poderia perfeitamente se perguntar: "Eu quero que as pessoas entendam isso e fiquem felizes ou eu quero que elas fiquem confusas e frustradas?". Se você escolher a segunda alternativa, há apenas um nível de simplicidade que assegurará seu sucesso: *extremamente simples*.

Seja consistente

Consistência é uma grande parte da simplicidade. Se você faz algo de um modo em um lugar, faço-o desse jeito em todo lugar.

Se você nomeia uma variável como `somethingLikeThis`, então todas as suas variáveis devem ser nomeadas desse jeito (`otherVariable`, `anotherNameLikeThat` etc.). Se você tem variáveis que são `nomeadas_desse_jeito`, então todas as variáveis devem ser em minúsculas e ter sublinhados entre as palavras.

Código que não é consistente é mais difícil para um programador entender e ler.

Podemos ilustrar isso observando um exemplo da linguagem natural. Compare essas duas frases:

- This is a normal sentence with normal words that everybody can understand.
(Essa é uma frase normal com palavras normais que todo mundo pode entender).
- tHisisanOrmalseNtencewitHnorMalwordsthAtevErybOdycAnunderStaNd.

Ambas as frases dizem exatamente a mesma coisa, mas a primeira é muito mais simples de ler porque é consistente com aquilo que a maioria das pessoas escreve em inglês. Claro, é possível ler a segunda frase, mas você desejaria ler um livro inteiro escrito desse jeito? Certo. Então, você desejaria ler um programa inteiro escrito sem nenhuma consistência?

Há situações em programação em que não importa como você faz as coisas, desde que você sempre as faça daquele jeito. Teoricamente, você poderia escrever seu código de algum modo maluco e complicado, mas, desde que fosse consistente com ele, as pessoas aprenderiam a lê-lo. (Claro, é melhor ser consistente e simples, mas se você não puder ser totalmente simples, pelo menos seja consistente).

Consistência total também pode tornar a programação mais fácil em muitos casos. Por exemplo, se todo objeto em seu programa tiver um campo chamado nome, você pode escrever um trecho de código simples que trata do campo nome de todo objeto em todo o seu programa. Mas, se no Objeto A, o campo nome é chamado um_nome e no Objeto B é chamado nome_meu, você terá que escrever um código especial para tratar do Objeto A e do Objeto B diferentemente.

De modo semelhante, seu programa deve se comportar de maneira consistente internamente. Um programador que está familiarizado com como usar uma parte de seu código deve estar imediatamente familiarizado com como usar outra parte de seu código, porque ambas as partes se comportam de maneira semelhante. Por exemplo, se, quando usar a Parte A, o programador tiver que chamar três funções e então escrever algum código, quando usar a Parte B, ele também deve ter que chamar um conjunto semelhante de três funções e então escrever algum código. E se você tiver uma função chamada dump na Parte A que faz a Parte A imprimir todas as suas variáveis internas, a função chamada dump na Parte B deve fazer a mesma coisa para a Parte B. Não obrigue os programadores a repreender o modo pelo qual seu sistema funciona toda vez que eles olham para um novo trecho dele.

Talvez as coisas não sejam tão consistentes no mundo real, mas você está encarregado do mundo de seu programa, então pode tornar as coisas simples e consistentes.

Há alguns exemplos de consistência no mundo real. Em grande parte da Ásia, as pessoas usam pauzinhos para comer. Nas Américas e na Europa, as pessoas usam garfos. Ok, são dois métodos diferentes de comer, mas, no geral, isso é muito consistente, em qualquer dada área. Agora, imagine se toda vez que você fosse à casa de alguém, tivesse que aprender algum modo totalmente novo de comer. Talvez na casa de Bob eles comam com tesouras e na casa de Mary com pedaços de papelão. Então, comer ficaria muito complexo, não?

É a mesma coisa em programação – sem consistência, as coisas ficam muito complexas. Com consistência, elas se tornam simples. E, mesmo que não sejam simples, pelo menos você pode aprender a complexidade apenas uma vez, e então você a sabe para sempre.

Legibilidade

Conforme foi dito muitas vezes no mundo do desenvolvimento de software, o código é lido com muito mais frequência do que é escrito. Então, é importante tornar o código fácil de ler:

Legibilidade de código depende primordialmente de como o espaço é ocupado por letras e símbolos.

Se todo o universo fosse preto, você não seria capaz de distinguir os objetos. Eles seriam uma única massa preta. Do mesmo modo, se um arquivo inteiro é uma massa de código sem o suficiente espaçamento consistente e lógico, é difícil separar as partes. Espaço é o que mantém as coisas separadas.

Você não quer espaço demais, porque então é difícil dizer como as coisas estão relacionadas.

E você não quer pouco demais, porque então é difícil dizer que as coisas estão separadas.

Não há nenhuma regra rígida sobre exatamente como o código deve ser espaçado, exceto que deve ser feito de maneira consistente e o espaçamento deve ajudar a informar o leitor sobre a estrutura do código.

Exemplo: Espaços

Este código é difícil de ler porque tem pouco espaço – pouca informação é dada sobre a estrutura do código:

```
x=1+2;y=3+4;z=x+y;if(z>y+x){print"error";}
```

Aqui está o mesmo bloco de código com espaço demais – o espaço impede o leitor de ver a estrutura do código:

```
x      =      1+      2;
y = 3      +4;
z = x      +  y;
if (z > y+x)
{   print "error";
}
```

Esse é ainda mais difícil de ler do que o código sem nenhum espaço.

Aqui está o mesmo código com razoável espaçamento:

```
x = 1 + 2;
y = 3 + 4;
z = x + y;
if (z > y + x) {
    print "error";
}
```

Esse é muito mais fácil de ler e o ajuda a perceber como o programador pretendia que o programa fosse projetado. Três variáveis são definidas e, então, conforme uma condição, um erro é exibido. Essa é a estrutura do programa, tornada clara ao leitor pelo jeito como o programador usou o espaço.

Tornar o código fácil de ler também ajuda a torná-lo fácil de corrigir. No exemplo anterior, quando o código é adequadamente espaçado, podemos ver facilmente que z nunca será maior do que y + x, porque z é sempre igual a y + x. Assim, o bloco iniciado com if (z > y + x) deve ser eliminado, uma vez que é desnecessário.

Em geral, se você tem algum código com muitos erros que também seja difícil de ler, a primeira coisa que você deve fazer é torná-lo mais legível. Assim você poderá ver com mais facilidade onde os erros estão.

Nomeando coisas

Uma parte importante da legibilidade é dar nomes adequados a variáveis, funções, classes etc. Idealmente:

Nomes devem ser longos o bastante para comunicar plenamente o que algo é ou faz sem serem tão longos que se tornem difíceis de ler.

Também é importante pensar em como a função, variável etc. vai ser usada. Uma vez que começamos a colocar seu nome em linhas de código, ele tornará essas linhas de código tão longas que são difíceis de ler? Por exemplo, se você tem uma função que é chamada apenas uma vez, em uma linha sozinha (sem nenhum código nessa linha), ela pode ter um nome razoavelmente longo. Contudo, uma função que você vai usar com frequência em expressões complexas deve provavelmente ter um nome que seja curto (embora ainda longo o bastante para comunicar plenamente o que ela faz).

Exemplo: Nomes

Aqui está um código com nomes inadequados:

```
q = s(j, f, m);
p(q);
```

Esses nomes não comunicam o que as variáveis são ou o que as funções fazem.

Aqui está o mesmo código com nomes adequados:

```
quarterly_total = sum(january, february, march);
print(quarterly_total);
```

E aqui está o mesmo código, com nomes tão longos que são difíceis de ler:

```
quarterly_total_for_company_in_2011_as_of_today =
add_all_of_these_together_and_return_the_result(january_total_amount,
february_total_amount, march_total_amount);
send_to_screen_and_dont_wait_for_user_to_respond(quarterly_total_for_
company_in_2011_as_of_today);
```

Esses nomes ocupam muito espaço, o que os torna difíceis de ler. Assim, de certa forma, nomear coisas também tem a ver com como letras e símbolos ocupam espaço.

Comentários

Ter bons comentários no código é uma grande parte de torná-lo legível. Contudo, você geralmente não deve inserir comentários que dizem o que um trecho de código está fazendo. Isso dever ser óbvio a partir da leitura do código. Se isso não é óbvio, o código deve ser tornado mais simples. Apenas se você não puder tornar o código mais simples, deverá ter um comentário explicando o que ele faz.

O verdadeiro propósito dos comentários é explicar por que você fez algo, quando a razão não é óbvia. Se você não explicar isso, outros programadores poderão ficar confusos e, quando eles forem alterar seu código, poderão eliminar partes importantes dele se essas partes não parecerem ter uma razão para existir.

Algumas pessoas acreditam que a legibilidade é o fator essencial da simplicidade do código – se seu código for fácil de ler, você fez tudo que precisa fazer como um projetista. Isso não é verdade – você pode ter código muito legível e ainda ter um sistema complexo demais. Contudo, tornar seu código legível é muito importante, e é normalmente o primeiro passo que deve ser dado na estrada rumo ao bom projeto de software.

Simplicidade exige projeto

Infelizmente, as pessoas não criam naturalmente sistemas simples. Sem atenção dada ao projeto, um sistema evoluirá para alguma coisa bem complicada.

Se seu projeto não é um bom projeto, e ele continua a crescer, você acabará tendo complexidade em demasia. Isso é difícil de imaginar para certas pessoas – algumas não conseguem imaginar que há um futuro além do almoço, e outras simplesmente não têm experiência suficiente para entender quão complexas as coisas podem ficar. E pode haver uma cultura corporativa que diz: “Oh, apenas incluímos ilegalmente alguns recursos novos; deveríamos fazer as coisas da maneira certa, mas não podemos porque blá-blá-blá”. Mas, um dia, seu projeto fracassará. E não importa quantas razões você possa dar para esse fracasso, isso não mudará o fato de que seu projeto fracassou.

No outro lado das coisas, quando você projetou bem, com frequência não há muito crédito vindo ao seu encontro. Fracassos catastróficos em projeto são grandes e perceptíveis, ao passo que pequenos incrementos de trabalho visando a um bom projeto não são visíveis pelas pessoas que não estão intimamente ligadas ao código. Isso pode fazer com que ser um projetista seja um trabalho difícil. Administrar um grande fracasso lhe traz muito reconhecimento, mas impedir que um jamais aconteça... bem, é provável que ninguém perceba.

Então, vamos parabenizá-lo aqui. Você pensou um pouco em projeto? Ótimo! Seus usuários e colegas desenvolvedores verão os benefícios – software funcional, lançamentos de versões pontuais e um código-base claro e compreensível. Você se sentirá confiante em seu próprio trabalho e irá para casa sentindo-se realizado. Será que os outros desenvolvedores saberão quanto trabalho foi necessário para fazer as coisas funcionarem tão bem? Talvez não. Mas tudo bem. Há outras recompensas no mundo além dos parabéns de seus pares.

Às vezes, contudo, você receberá algum reconhecimento por todo o seu trabalho. Não se desespere – alguém acabará notando. E, até então, desfrute de todos os outros resultados positivos do projeto eficaz e correto.



Quando você começar a aplicar os princípios de projeto deste livro ao seu projeto, poderá levar muito tempo para que seus programadores juniores ou colegas entendam por que eles também devem projetar bem. Fazer com que eles leiam este livro ajudará. Se eles não puderem ou não quiserem lê-lo, continue orientando-os (ou forçando-os, na pior das hipóteses) para boas decisões de projeto e eles verão após alguns anos (do lado de fora) como boas decisões de projeto compensam.

CAPÍTULO 8

Complexidade

Quando você trabalha como um programador profissional, há chances de que conhecerá alguém (ou de que você seja alguém!) que está passando por essa história de horror do desenvolvimento comum: “Começamos a trabalhar neste projeto cinco anos atrás e a tecnologia que estávamos usando/fazendo era moderna na época, mas está obsoleta agora. As coisas estão ficando cada vez mais complexas com essa tecnologia obsoleta, então, torna-se cada vez menos provável que algum dia terminaremos o projeto. Mas, se reescrevermos, poderíamos estar aqui por mais cinco anos!”.

Outra popular é: “Não podemos desenvolver rápido o suficiente para acompanhar as necessidades do usuário moderno”. Ou: “Enquanto estamos desenvolvendo, a Empresa X escreveu um produto melhor que o nosso muito mais rápido do que nós”.

Sabemos agora que a fonte desses problemas é a complexidade. Você inicia com um projeto simples que pode ser concluído em um mês. Então você adiciona complexidade, e a tarefa levará três meses. Então você pega cada parte dele e a torna mais complexa, e a tarefa levará nove meses.

Complexidade fundamenta-se em complexidade – não é apenas uma coisa linear. Isto é, você não pode fazer suposições como: “Temos dez recursos, então, adicionar mais um adicionará apenas 10% mais de tempo”. De fato, esse novo recurso terá que ser coordenado com todos os dez dos seus recursos existentes. Então, se leva dez horas de tempo de codificação para implementar o recurso em si, pode perfeitamente levar mais dez horas de tempo de codificação para fazer com que os dez recursos existentes interajam todos adequadamente com o novo recurso. Quanto

mais recursos há, maior fica o custo de adicionar um recurso. Você pode minimizar esse problema tendo um excelente projeto de software, mas sempre haverá um pequeno custo extra para todo novo recurso.

Alguns projetos começam com um conjunto de requisitos tão complexo que eles nunca chegam à sua primeira versão. Se você está nessa situação, deve apenas retocar os recursos. Não aposte alto no seu primeiro lançamento – lance algo que funcione e faça-o funcionar melhor com o tempo.

Há outros modos de adicionar complexidade do que apenas adicionar recursos, também. Os outros modos mais comuns são:

Expandir o propósito do software

No geral, simplesmente nunca faça isso. Seu departamento de marketing talvez estivesse com a ideia de fazer um único software que calcule seus impostos e cozine o jantar, mas você deve estar gritando o mais alto que pode sempre que qualquer sugestão como essa chega perto da sua mesa. Atenha-se ao propósito existente de seu software – ele apenas tem de fazer o que faz bem, e você será bem-sucedido (desde que seu software ajude as pessoas com algo que elas realmente precisem e queiram).

Acrescentar programadores

Sim, está certo – acrescentar mais pessoas à equipe não torna as coisas mais simples; em vez disso, adiciona complexidade. Há um famoso livro chamado “O Mítico Homem-Mês”, de Fred Brooks, que evidencia isso. Se você tem dez programadores, acrescentar um décimo primeiro significa gastar tempo para entrosar esse programador, mais o tempo para entrosar os dez programadores com a nova pessoa, mais o tempo gasto pela nova pessoa interagindo com os dez programadores, e assim por diante. Você tem mais probabilidade de ser bem-sucedido com um pequeno grupo de programadores especializados do que com um grande grupo de programadores não especializados.

Alterar coisas que não precisam ser alteradas

A qualquer momento que você altera algo, você está adicionando complexidade. Se é um requisito, um projeto ou apenas um trecho de código, você está introduzindo a possibilidade de erros, bem como o tempo exigido para decidir sobre a alteração, o tempo exigido para

implementar a alteração, o tempo exigido para validar o funcionamento da nova alteração com todas as outras partes do software, o tempo exigido para acompanhar a alteração e o tempo exigido para testar a alteração. Cada alteração fundamenta-se na última em termos de toda essa complexidade, então, quanto mais você altera, mais tempo cada nova alteração vai levar. Ainda é importante fazer certas alterações, mas você deve estar tomando decisões com base em informações sobre elas, não apenas fazendo alterações por capricho.

Ficar preso a tecnologias ruins

Basicamente, é aqui que você decide usar alguma tecnologia e então fica preso a ela durante muito tempo porque você está tão dependente dela. Uma tecnologia nesse sentido é “ruim” se ela o prende (não permite que você mude facilmente para alguma outra tecnologia no futuro), não vai ser flexível o suficiente para suas necessidades futuras ou simplesmente não tem o nível de qualidade que você precisa para projetar softwares simples com ela.

Mal-entendido

Programadores que não entendem plenamente seu trabalho tendem a desenvolver sistemas complexos. Pode tornar-se um ciclo vicioso: mal-entendido leva à complexidade, o que leva a mais mal-entendido, e assim por diante. Um dos melhores modos de melhorar suas habilidades de projeto é estar certo de que você entende plenamente os sistemas e ferramentas com que está trabalhando. Quanto melhor você os entende e quanto mais você sabe sobre software em geral, mais simples seus projetos podem ser.

Projeto ruim ou nenhum projeto

Basicamente, isso apenas significa “um fracasso em planejar uma alteração”. As coisas vão mudar, e exige-se trabalho de projeto para manter a simplicidade enquanto o projeto cresce. Você tem que projetar bem desde o início e continuar projetando bem enquanto o sistema se expande – caso contrário, você pode introduzir grande complexidade muito rápido, porque com um projeto ruim, cada novo recurso multiplica a complexidade do código em vez de apenas adicionar um pouquinho a ele.

Reinventando a roda

Se, por exemplo, você criar seu próprio protocolo quando um bom existe, você vai passar muito tempo trabalhando no protocolo, quando poderia apenas estar trabalhando em seu software. Você quase nunca deve ter qualquer dependência interna inventada, como um web server, um protocolo ou uma biblioteca importante, a menos que esse seja seu produto. As únicas vezes em que se pode dizer que reinventar a roda é aceitável é quando quaisquer dos seguintes pontos forem verdadeiros:

- a. Você precisa de algo que ainda não existe.
- b. Todas as “rodas” existentes são tecnologias ruins que o prenderão.
- c. As “rodas” existentes são incapazes de lidar com suas necessidades.
- d. As “rodas” existentes não estão sendo adequadamente submetidas à manutenção e você não pode assumir a manutenção delas (porque, por exemplo, você não tem o código-fonte).

Todos esses fatores são lenta e gradualmente prejudiciais a seu projeto, não imediatamente destrutivos. A maioria deles apenas causa danos de longo prazo – algo que você não verá durante um ano ou mais – então, quando alguém os propõe, com frequência eles parecem inofensivos. E, mesmo quando você começa a implementá-los, eles podem parecer ótimos. Mas, com o passar do tempo – e particularmente enquanto cada vez mais desses se acumulam –, a complexidade torna-se mais aparente e cresce, e cresce, e cresce, até que você seja outra vítima dessa história de horror cada vez mais comum, O Produto Que Nunca É Despachado.

Complexidade e propósito

O propósito básico de qualquer sistema no qual você está trabalhando deveria ser bem simples. Isso ajuda a manter o sistema como um todo tão simples quanto ele possa realisticamente ser. Mas, se você começar a adicionar recursos que cumprem algum outro propósito, as coisas ficarão muito complexas rapidamente. Por exemplo, o propósito básico de um editor de texto é ajudar você a escrever coisas. Se de repente o tornássemos também capaz de ler seus e-mails, ele ficaria complicado.

Você pode imaginar como pareceria a interface do usuário? Onde você colocaria todos os botões? Diríamos que essa é uma violação do propósito do editor de texto. Você nem mesmo expandiu o propósito dele; você apenas adicionou recursos que nada têm nada a ver com ele.

Também é importante pensar no propósito do usuário. Seu usuário estará tentando fazer algo. Idealmente, o propósito de um programa deveria estar muito próximo (nas palavras exatas que você usaria para descrevê-lo) do propósito do usuário. Por exemplo, digamos que o propósito do usuário é fazer seus impostos. Ele quer software cujo propósito é ajudar as pessoas a fazer seus impostos.

Se seu propósito e o propósito do usuário não combinam, você provavelmente está dificultando sua vida. Por exemplo, se ele quer ler os emails, mas o propósito básico do programa que está usando é mostrar anúncios aos usuários, esses propósitos não estão combinados.

Quer ver seu usuário ficar zangado rapidamente? Torne difícil para ele alcançar seu propósito. Faça com que janelas apareçam em seu rosto quando ele estiver tentando fazer algo. Adicione tantos recursos ao seu programa que ele não consegue encontrar o certo. Use muitos ícones estranhos que ele não consegue entender. Há muitos modos de fazer isso, mas todos eles se resumem a interferir no propósito do usuário ou violar o propósito básico do próprio programa.

Às vezes, marqueteiros ou gerentes têm metas para um programa que não estão realmente alinhadas com o propósito básico do programa, como “ser bonitinho”, “ter um projeto vantajoso”, “tornar-se popular com a mídia de notícias”, “usar as tecnologias mais recentes” e assim por diante. Essas pessoas podem ser importantes para sua organização, mas elas não são as pessoas que devem decidir o que seu programa faz! Como projetista de software ou gerente técnico, é seu trabalho cuidar para que o programa mantenha o curso e nunca viole seu propósito básico. Ninguém mais vai assumir essa responsabilidade. Às vezes, você poderia realmente ter que lutar por isso, mas vale muito a pena no longo prazo.

E não é como se você tivesse chegado a um fracasso de marketing com essa filosofia. Há muitos, muitos produtos que foram extremamente bem-sucedidos, atendo-se a apenas um propósito. O propósito do sabão

é apenas limpar as coisas. O sal apenas torna as coisas salgadas. Uma lâmpada apenas ilumina as coisas. Mas todos esses são produtos que sustentam enormes corporações há décadas. Você não precisa ter um produto complicado para ter marketing eficaz – você só precisa ter conhecimento e habilidade em marketing, que é uma área completamente separada de projeto de software.

Realmente, não há necessidade de tornar-se sofisticado e complexo e tentar fazer 500 coisas ao mesmo tempo em um único programa. Os usuários ficam muito felizes com um produto focado e simples que nunca viola seu propósito básico.

Tecnologias ruins

Outra fonte comum de complexidade é selecionar a tecnologia errada para usar em seu sistema – particularmente uma que acabará não resistindo bem a futuros requisitos. Contudo, pode ser complicado saber, sem ser capaz de prever o futuro, qual tecnologia você deve escolher agora. Felizmente, há três fatores que você pode observar para determinar se uma tecnologia é “ruim” antes mesmo de começar a usá-la: potencial de sobrevivência, interoperabilidade e atenção à qualidade.

Potencial de sobrevivência

O potencial de sobrevivência de uma tecnologia é a probabilidade de que ela continuará a ser submetida à manutenção. Se você ficar preso a uma biblioteca ou alguma dependência que se torna obsoleta e não sujeita à manutenção, você está prestes a ter algum problema.

Você pode ter alguma ideia do potencial de sobrevivência de um software observando o seu histórico recente de lançamento de versões. Os desenvolvedores têm lançado com frequência novas versões que resolvem os verdadeiros problemas do usuário? Além disso, com que intensidade os desenvolvedores reagem a relatos de erros? Eles têm uma lista de discussão ou uma equipe de suporte que é muito ativa? Há muitas pessoas online falando sobre essa tecnologia? Se uma tecnologia tem muita força agora, você pode ter razoável certeza de que ela não vai morrer tão cedo.

Observe também se um vendedor está promovendo a tecnologia, ou se ela é amplamente aceita e usada em muitas áreas de software por muitos desenvolvedores diferentes. Se há apenas um vendedor que promove e encaminha o sistema, há um risco de que esse vendedor ou saia do ramo ou apenas decida parar de oferecer manutenção ao sistema.

Popularidade

Pode parecer que estamos dizendo que você deve simplesmente selecionar a tecnologia mais popular que satisfaz suas necessidades. Até certo ponto, isso é verdade – tecnologias populares têm muito potencial de sobrevivência. Contudo, você tem que observar a diferença entre ferramentas que são validamente populares e ferramentas que são populares apenas porque detêm algum tipo de monopólio.

No momento da redação deste livro, C é um exemplo de linguagem validamente popular. Muitas pessoas a usam em muitas organizações diferentes para muitos propósitos diferentes. Ela é o assunto de vários padrões internacionais, e há numerosas implementações desses padrões, incluindo muitos compiladores diferentes largamente usados.

Algumas tecnologias são populares apenas porque você deve usá-las, contudo. Suponha que a Empresa X projete sua própria linguagem de programação. Então ela projeta um dispositivo popular que aceita apenas programas escritos naquela linguagem. Esse é o caso de “um vendedor” mencionado no texto – a linguagem pode parecer popular, mas na verdade tem baixo potencial de sobrevivência, a menos que seja adotada amplamente pela indústria de software.

Interoperabilidade

Interoperabilidade é uma medida de quanto fácil é afastar-se de uma tecnologia se você tiver que fazer isso. Para ter uma ideia da interoperabilidade de uma tecnologia, pergunte-se: “Podemos interagir com essa tecnologia de algum modo-padrão, para que seja fácil mudar para outro sistema que segue o mesmo padrão?”.

Por exemplo, há padrões internacionais para como um programa deve interagir com um sistema de banco de dados. Alguns sistemas de bancos de dados suportam essas normas muito bem. Se você pegar um desses

bons sistemas de bancos de dados, poderá mudar para outro sistema de banco de dados no futuro com pequenas alterações em seu programa. Contudo, alguns outros sistemas de bancos de dados não são muito bons em suportar padrões. Se você quiser mudar entre sistemas de bancos de dados que não suportam padrões, terá que reescrever seu programa. Então, quando escolher um desses sistemas não padronizados, você ficará preso a ele e será incapaz de mudar facilmente para um sistema diferente.

Atenção à qualidade

Essa é a medida mais subjetiva, mas a ideia é ver se o produto tem melhorado em suas recentes versões. Se você puder ver o código-fonte, verifique se os desenvolvedores estão refatorando e limpando a base de código. Está ficando mais fácil ou mais complexo usar? As pessoas que fazem a manutenção da tecnologia realmente se preocupam com a qualidade de seu produto? Houve recentemente muitas vulnerabilidades sérias de segurança no software que parecem ter sido resultado de programação deficiente?

Outras razões

Há outros aspectos a considerar quando você está escolhendo uma tecnologia – principalmente sua simplicidade e quanto adequada ela é para seus propósitos. Opinião pessoal pode desempenhar um papel, também, depois que você levou em conta todas as considerações práticas. Algumas pessoas gostam do jeito de uma linguagem de programação parecer melhor do que outra. Isso pode às vezes ser uma razão válida para escolher uma tecnologia – se você simplesmente gosta mais de uma tecnologia do que de outra e tudo o mais é igual entre elas, fique com a que o faz feliz. Afinal de contas, você é quem irá usá-la – sua opinião importa! As diretrizes acima o ajudarão a eliminar as escolhas definitivamente ruins; o restante cabe à sua pesquisa, requisitos e desejos pessoais.

Complexidade e a solução errada

Com frequência, se algo está ficando muito complexo, isso significa que há um erro no projeto em algum lugar bem abaixo do nível onde a complexidade aparece.

Por exemplo, é muito difícil fazer um carro correr rápido se ele tiver rodas quadradas. Ajustar o motor não vai resolver o problema – você precisa reprojetar o carro para que suas rodas fiquem redondas.

Qualquer hora que haja uma “complexidade insolúvel” em seu programa, é porque há algo fundamentalmente errado com o projeto. Se o problema parece insolúvel em um nível, recue e observe o que pode estar subjacente ao problema.

Programadores realmente fazem isso com muita frequência. Você pode pensar: “Tenho um código terrivelmente confuso e é realmente complexo adicionar um novo recurso”? Bem, seu problema fundamental é que o código está confuso. Limpe-o, torne o código já existente simples e você descobrirá que adicionar o novo recurso será simples também.

Qual problema você está tentando resolver?

Se alguém chega para você e diz algo como: “Como faço esse pônei voar até a lua?”, a pergunta que você precisa fazer é: “Qual problema você está tentando resolver?”. Talvez você descubra que o que essa pessoa realmente precisa é coletar algumas rochas cinzentas. Por que ela pensou que tinha que voar até a lua e usar um pônei para isso, apenas ela pode saber. As pessoas realmente ficam confusas assim. Pergunte a elas qual problema estão tentando resolver e uma solução simples começará a surgir. Por exemplo, nesse caso, uma vez que entendemos o problema plenamente, a solução se torna simples e óbvia: a pessoa deve apenas sair e encontrar algumas rochas cinzentas – nenhum pônei é exigido.

Então, quando as coisas ficarem complexas, recue e dê uma olhada no problema que você está tentando resolver. Dê um passo realmente grande para trás. Você tem permissão para questionar tudo. Talvez você tenha pensado que adicionar dois e dois era o único modo de obter quatro, e você não pensou em adicionar um e três ao invés, ou saltar a

adição inteiramente e simplesmente colocar quatro lá. O problema é: “Como obtenho o número quatro?”. Qualquer método para resolver esse problema é aceitável, então, o que você precisa é entender qual seria o melhor método para a situação em que você se encontra.

Descarte suas suposições. Realmente olhe para o problema que você está tentando resolver. Certifique-se de que você entende plenamente todos os aspectos dele e então descubra o modo mais simples de resolvê-lo. Não pergunte: “Como resolver esse problema usando meu código atual?” ou “Como a Professora Anne resolveu esse problema em seu programa?”. Não – apenas pergunte-se: “Como, em geral, em um mundo perfeito, esse tipo de problema deve ser resolvido?”. A partir daí, talvez você veja como seu código precisa ser retrabalhado. Então você poderá retrabalhar seu código e resolver o problema.

Problemas complexos

Às vezes você será chamado para resolver um problema que é inherentemente muito complexo – por exemplo, correção ortográfica ou fazer um computador jogar xadrez. Isso não significa que sua solução tem que ser complexa, mas significa que você terá que trabalhar mais do que o normal para simplificar seu código quando tratar desse problema.

Se você está tendo dificuldade com um problema complexo, anote-o em um papel em linguagem simples ou desenhe-o na forma de um diagrama. Realmente, muito da melhor programação é feita em papel. Colocá-lo no computador é apenas um detalhe.

A maioria dos problemas difíceis de projeto pode ser resolvida simplesmente desenhando ou escrevendo em um papel.

Lidando com a complexidade

Como programador, você se deparará com a complexidade. Outros programadores escreverão programas complexos que você terá que corrigir. Projetistas de hardware e projetistas de linguagens tornarão sua vida difícil.

Se alguma parte de seu sistema é complexa demais, há um modo específico de corrigi-la – reprojete as partes individuais, em pequenos passos. Cada correção deve ser tão pequena quanto você pode fazê-la com segurança sem introduzir complexidade adicional. Quando você está passando por esse processo, o maior perigo é que poderia possivelmente introduzir mais complexidade com suas correções. É por isso que tantos reprojetos ou reescritas definitivamente falham – eles introduzem mais complexidade do que corrigem, ou acabam sendo apenas tão complexos quanto era o sistema original.

Cada passo pode ser tão pequeno quanto dar a uma única variável um nome melhor, ou apenas adicionar alguns comentários a um código confuso. Mas, com mais frequência, os passos envolvem dividir uma parte complexa em várias partes simples. Por exemplo, se você tem um arquivo extenso que contém todo o seu código, comece a melhorá-lo dividindo uma parte pequena em um arquivo separado. Então melhore o projeto dessa parte pequena. Então divida alguma outra parte pequena do sistema em um novo arquivo e melhore seu projeto. Continue assim e posteriormente você terminará com um sistema confiável, compreensível e sujeito à manutenção.

Se seu sistema é muito complexo, isso pode exigir muito trabalho, então, você deve ser paciente. Você deve primeiro conceber um sistema que seja mais simples do que aquele que tem agora – mesmo que seja apenas de um jeito modesto. Então você trabalha pensando nesse sistema mais simples, passo a passo. Uma vez que você alcance esse sistema mais simples, concebe novamente um sistema ainda mais simples e trabalha nele. Você nem sempre tem que conceber o sistema “perfeito”, porque isso não existe. Você apenas precisa trabalhar continuamente com a intenção de chegar um sistema que seja melhor do que aquele que tem agora e, posteriormente, você alcançará um nível de simplicidade administrável.

É importante notar que você não pode parar de escrever recursos e passar muito tempo apenas reprojetando. A Lei da Alteração nos diz que o ambiente em torno de seu programa estará continuamente mudando, e assim a funcionalidade de seu programa deve se adaptar. Se você deixar de se adaptar e melhorar sob o ponto de vista do usuário durante qualquer período significativo de tempo, corre o risco de perda de sua base de usuários e de morte de seu projeto.

Há, felizmente, diversos modos de equilibrar essas duas necessidades de escrever recursos e lidar com a complexidade. Um dos melhores modos é fazer seu reprojeto com o objetivo de tornar algum recurso específico mais fácil de implementar, e então implementá-lo. Desse modo, você alterna regularmente entre trabalho de reprojeto e trabalho de recursos. Isso também ajuda seu novo projeto a se adequar bem às suas necessidades, porque você o está criando com um uso real em mente. Seu sistema ficará aos poucos menos complexo com o tempo e você ainda acompanhará as necessidades de seus usuários. Você pode até fazer isso para erros – se perceber que algum erro será mais fácil de corrigir com um projeto diferente, reprojete o código antes de corrigi-lo.

Reprojetando para um recurso

Um projeto chamado Bugzilla armazena todos os seus dados em um banco de dados. Bugzilla suporta apenas um sistema de banco de dados em particular para armazenar dados, chamado OldDB. Alguns clientes novos querem usar um sistema de banco de dados diferente para armazenar dados, chamado NewDB. Esses clientes têm boas razões para querer esse recurso: eles entendem que NewDB é muito melhor do que OldDB, e eles já têm NewDB sendo executado em suas empresas. Mas todos os clientes existentes querem continuar usando OldDB.

Então, Bugzilla tem que começar a suportar mais de um banco de dados. Isso exigirá muitas alterações de código, já que Bugzilla não tem nenhum código centralizado para armazenar e receber informações do banco de dados. Em vez disso, há muitos comandos customizados de banco de dados espalhados por todo o código, que são específicos do OldDB e não funcionarão no NewDB.

Uma opção é inserir comandos if por toda a base de código, escrevendo código diferente para NewDB e OldDB em todo lugar em que o banco de dados é acessado. Isso praticamente dobraria a complexidade de toda a base de código, contudo, e a equipe do Bugzilla consiste de apenas alguns programadores em tempo parcial. Se a complexidade do sistema dobrasse, eles não poderiam mais fazer a manutenção dele.

Em vez disso, a equipe do Bugzilla decide reprojetar o sistema para que ele possa suportar múltiplos bancos de dados. Esse é um projeto enorme. Aqui está uma visão geral de alto nível de como eles o executam:

Reescrevendo

Alguns projetistas, quando confrontados com um sistema muito complexo, jogam-no fora e começam tudo de novo. Contudo, reescrever um sistema a partir do zero é essencialmente uma admissão de fracasso como projetista. É fazer a declaração: “Fracassamos em projetar um sistema sujeito à manutenção e então devemos começar novamente”.

Algumas pessoas acreditam que todos os sistemas devem posteriormente ser reescritos. Isso não é verdade. É possível projetar um sistema que nunca precise ser jogado fora. Um projetista de software dizer: “Teremos que jogar a coisa toda fora algum dia de qualquer forma” seria o mesmo que um arquiteto dizer: “Esse arranha-céu irá desabar algum dia de qualquer forma”. Se o arranha-céu fosse deficientemente projetado e não recebesse boa manutenção, então, sim, algum dia ele desabaria. Mas se ele fosse construído de maneira correta desde o início e então recebesse manutenção adequada, por que ele desabaria?

É igualmente possível desenvolver sistemas de software sujeitos à manutenção como é construir arranha-céus robustos.

Agora, com tudo isso dito, há situações em que reescrever é aceitável. Contudo, elas são muito raras. Você deve apenas reescrever se todos os seguintes pontos forem verdadeiros:

1. Você desenvolveu uma estimativa precisa que mostra que reescrever o sistema será um uso mais eficiente do tempo do que reprojetar o sistema existente. Não suponha apenas – faça experimentos reais com o reprojeto do sistema existente para ver como funciona. Pode ser muito difícil confrontar a complexidade existente e resolver alguma parte dela, mas você deve realmente tentar isso algumas vezes antes de poder saber quanto esforço para corrigir tudo será necessário.
2. Você tem muito tempo para passar criando um novo sistema.
3. Você é de algum modo um projetista melhor do que o projetista original do sistema ou, se você é o projetista original, suas habilidades de projeto melhoraram drasticamente desde que você projetou o sistema original.

4. Você pretende projetar esse novo sistema em uma série de passos simples e ter usuários que podem dar-lhe feedback para cada passo ao longo do caminho.
5. Você tem os recursos disponíveis tanto para fazer manutenção no sistema existente quanto para projetar um novo ao mesmo tempo. Nunca pare de dar manutenção em um sistema que está em uso para que os programadores possam reescrevê-lo. Os sistemas devem sempre estar sujeitos à manutenção se estiverem em uso. E lembre-se que sua atenção pessoal também é um recurso que deve ser levado em conta aqui – você tem tempo suficiente disponível diariamente para ser um projetista tanto no novo sistema quanto no velho sistema simultaneamente, se for trabalhar em ambos?

Se todos os pontos anteriores são verdadeiros, você pode estar em uma situação em que é aceitável reescrever. Caso contrário, a coisa correta a fazer é lidar com a complexidade do sistema existente sem reescrever, melhorando o projeto do sistema em uma série de passos simples.

CAPÍTULO 9

Testes

Não há certeza de que um programa será executado no futuro – há apenas a certeza de que um programa está sendo executado agora. Mesmo que o tenha executado uma vez, pode não ser executado novamente. Talvez o ambiente em torno de si mude, de modo que não funcione mais. Talvez você execute em um computador diferente e ele não funcione na nova máquina.

Contudo, há esperança – não estamos fadados à incerteza infinita sobre a funcionalidade de nosso software. A Lei de Testes nos diz a saída:

O grau em que você sabe como seu software se comporta é o grau em que você o testou com precisão.

Quanto mais recentemente você testou seu software, mais provável é que ele ainda funcione. Em quanto mais ambientes você o testou, mais certo pode estar de que ele funciona nessas circunstâncias. Isso é parte do que queremos dizer quando falamos do “grau” dos testes – quantos aspectos do software você testou, quão recentemente, e em quantos ambientes diferentes. Em geral, você poderia simplesmente dizer:

A menos que tenha tentado, você não sabe que ele funciona.

Dizer “ele funciona” é realmente muito vago, contudo – o que você quer dizer com “funciona”? O que você realmente sabe quando testa é que seu software se comporta como pretendia que ele se comportasse. Assim, você tem que saber qual comportamento você pretendia. Isso pode parecer óbvio, mas é um fato crítico em testes. Você deve fazer uma pergunta muito precisa com todo teste e obter uma resposta muito específica. A pergunta poderia ser algo como: “O que acontece quando um usuário pressiona esse botão como a primeira coisa que ele faz depois que a

aplicação inicia, quando a aplicação nunca foi iniciada antes?”. E você deve estar procurando alguma resposta específica, tal como: “A aplicação exige uma janela que diz: ‘Alô, mundo!’”.

Então, você tem uma pergunta, e você sabe qual deveria ser a resposta. Se você obtiver alguma outra resposta, então seu software “não está funcionando”.

Às vezes, é muito difícil avaliar um comportamento, e você pode apenas perguntar: “Se um usuário faz isso, o programa trava?”. E esperar a resposta: “Não”. Mas, com software bem projetado, na maioria das situações, você pode obter informações muito mais específicas do que aquelas com seus testes.



Você também deve tornar seus testes precisos. Se eles lhe dizem que seu programa está se comportando adequadamente quando não está – ou lhe dizem que está com problemas quando na verdade está funcionando bem – eles são testes imprecisos.

Finalmente, você deve observar os resultados de seus testes a fim de que eles sejam válidos. Se eles falharem, deve haver algum modo de você saber que falharam e especificamente como falharam.

Os testes podem ser facilmente desprezados. Escrevemos algum código, o salvamos e esquecemos de alguma vez ver se ele realmente funciona. Mas, não importa quão brilhante programador você é, não importa quantas provas matemáticas você faça para mostrar que seu código está correto, você não sabe que ele funciona a menos que tenha realmente tentado usá-lo.

E se, a qualquer altura, você alterar um trecho de seu software, você não sabe mais que esse trecho funciona. Ele deve ser testado novamente. Além disso, esse trecho provavelmente está relacionado a muitos outros trechos, então agora você não sabe se qualquer desses trechos funciona, também. Se sua alteração for suficientemente grande, talvez você tenha que testar todo o programa novamente.

Obviamente, você não quer testar manualmente todo o seu programa sempre que fizer uma pequena alteração. Então, nos tempos modernos, os desenvolvedores normalmente aplicam essa lei, criando testes automatizados para todo trecho de código que escrevem. A coisa boa disso

é que eles podem simplesmente executar os testes logo após terem feito qualquer alteração, e esses testes automatizados testarão todas as partes individuais do sistema para se certificar de que tudo funciona após cada alteração individual.

Há muita informação na Internet e em livros sobre como escrever testes automatizados e testes em geral – é uma área muito bem coberta, e vale a pena ler a respeito. A Lei de Testes apenas explica por que devemos testar, quando testar e quais informações os testes estão realmente nos dando.

APÊNDICE A

As leis do projeto de software

Este apêndice resume todas as verdadeiras leis discutidas neste livro:

1. O propósito do software é ajudar as pessoas.
2. A Equação do Projeto de Software é:

$$D = \frac{V_n + V_f}{E_i + E_m}$$

onde:

D Representa a desejabilidade por uma alteração.

V_n Representa o valor agora.

V_f Representa o valor futuro.

E_i Representa o esforço de implementação.

E_m Representa o esforço de manutenção.

Esta é a lei principal do projeto de software. Com o passar do tempo, esta equação se reduz a:

$$D = \frac{V_f}{E_m}$$

A qual demonstra que é mais importante reduzir o esforço de manutenção do que reduzir o esforço de implementação.

3. A Lei da Alteração: Quanto mais tempo seu programa existe, mais provável é que qualquer parte dele terá de mudar.
4. A Lei da Probabilidade de Erros: A chance de introduzir um erro em seu programa é proporcional à quantidade de alterações que você faz nele.

5. A Lei da Simplicidade: A facilidade de manutenção de qualquer software é proporcional à simplicidade de suas partes individuais.
6. A Lei de Testes: O grau em que você sabe como seu software se comporta é o grau em que você o testou com precisão.

É isso. Muitos mais fatores e ideias foram discutidos neste livro, mas esses seis itens são as leis de projeto de software. Observe que, de todos esses, os mais importantes a ter em mente são o propósito do software, a forma reduzida da Equação do Projeto de Software e a Lei da Simplicidade.

Se você quisesse resumir os fatos mais importantes sobre projeto de software em duas frases simples, elas seriam:

- É mais importante reduzir o esforço de manutenção do que o de implementação.
- O esforço de manutenção é proporcional à complexidade do sistema.

Com apenas essas duas declarações e um entendimento do propósito do software, você poderia muito possivelmente revolucionar toda a ciência do projeto de software, contanto que você também entendesse que a complexidade do sistema realmente vem da complexidade de suas partes individuais.

APÊNDICE B

Fatos, leis, regras e definições

Este apêndice lista todo fato, lei, regra e definição importante abordados neste livro:

- *Fato:* A diferença entre um mau e um bom programador é o entendimento. Isto é, maus programadores não entendem o que estão fazendo, e bons programadores, sim.
- *Regra:* Um “bom programador” deve fazer tudo o que puder para tornar o que ele escreve o mais simples possível a outros programadores.
- *Definição:* Um programa é:
 1. Uma sequência de instruções dadas ao computador;
 2. As ações executadas por um computador como resultado de ter recebido instruções.
- *Definição:* Qualquer coisa que envolve a arquitetura de seu sistema de software ou as decisões técnicas que você toma enquanto cria o sistema se enquadra na categoria de “projeto de software”.
- *Fato:* Todos aqueles que escrevem software são projetistas.
- *Regra:* Projeto não é uma democracia. Decisões devem ser tomadas por indivíduos.
- *Fato:* Há leis de projeto de software, elas podem ser conhecidas e você pode conhecê-las. Elas são eternas, imutáveis e fundamentalmente verdadeiras, e funcionam.
- *Lei:* O propósito do software é ajudar as pessoas.
- *Fato:* As metas do projeto de software são:

1. Permitir-nos escrever software que seja o mais útil possível;
 2. Permitir que nosso software continue a ser o mais útil possível;
 3. Projetar sistemas que podem ser desenvolvidos e sujeitos à manutenção do modo mais fácil possível por seus programadores, para que eles possam ser – e continuem a ser – os mais úteis possíveis.
- **Lei:** A Equação do Projeto de Software:

$$D = \frac{V_n + V_f}{E_i + E_m}$$

Esta é a Lei Principal do Projeto de Software. Ou, em inglês:

A desejabilidade por uma alteração é diretamente proporcional ao valor de agora mais o valor futuro e inversamente proporcional ao esforço de implementação mais o esforço de manutenção.

Com o passar do tempo, esta equação se reduz a:

$$D = \frac{V_f}{E_m}$$

A qual demonstra que é mais importante reduzir o esforço de manutenção do que reduzir o esforço de implementação.

- **Regra:** O nível de qualidade de seu projeto deve ser proporcional à extensão de tempo futuro que seu sistema continuará a ajudar as pessoas.
- **Regra:** Há algumas coisas sobre o futuro que você não sabe.
- **Fato:** O erro mais comum e desastroso que programadores cometem é prever algo sobre o futuro quando de fato eles não podem saber.
- **Regra:** Você está mais seguro se não tenta prever o futuro e, ao invés, toma todas as suas decisões de projeto com base em informações atuais conhecidas.
- **Lei:** A Lei da Alteração: Quanto mais tempo seu programa existe, mais provável é que qualquer parte dele terá que ser alterada.
- **Fato:** Os três erros (chamados de “as três falhas” neste livro) que os projetistas de software estão propensos a cometer quando se confrontam com a Lei da Alteração são:

1. Escrever código que não é necessário;
 2. Não tornar o código fácil de alterar;
 3. Ser genérico demais.
- **Regra:** Não escreva código até que você realmente precise dele, e elimine qualquer código que não esteja sendo usado.
 - **Regra:** Código deve ser projetado com base no que você sabe agora, não no que você pensa que acontecerá no futuro.
 - **Fato:** Quando seu projeto realmente torna as coisas mais complexas em vez de simplificá-las, você está fazendo superengenharia.
 - **Regra:** Seja apenas tão genérico quanto sabe que precisa ser, imediatamente.
 - **Regra:** Você pode evitar as três falhas fazendo desenvolvimento e projeto incrementais.
 - **Lei:** A Lei da Probabilidade de Erros: A chance de introduzir um erro em seu programa é proporcional à extensão das alterações que você faz nele.
 - **Regra:** O melhor projeto é o que permite a maior alteração no ambiente com a menor alteração no software.
 - **Regra:** Nunca “corrija” qualquer coisa a menos que seja um problema e você tenha evidência de que o problema realmente existe.
 - **Regra:** Em qualquer sistema particular, qualquer informação deve, idealmente, existir apenas uma vez.
 - **Lei:** A Lei da Simplicidade: A facilidade de manutenção de qualquer software é proporcional à simplicidade de suas partes individuais.
 - **Fato:** Simplicidade é relativa.
 - **Regra:** Se você realmente quer ser bem-sucedido, é melhor ser extremamente simples.
 - **Regra:** Seja consistente.
 - **Regra:** A legibilidade do código depende principalmente de quanto espaço é ocupado por letras e símbolos.

- *Regra:* Nomes devem ser extensos o bastante para comunicar plenamente o que algo é ou faz sem ser tão extensos que se tornem difíceis de ler.
- *Regra:* Comentários devem explicar por que o código está fazendo algo, não o que ele está fazendo.
- *Regra:* Simplicidade exige projeto.
- *Regra:* Você pode criar complexidade:
 - Expandido o propósito de seu software;
 - Adicionando programadores à sua equipe;
 - Alterando coisas que não precisam ser alteradas;
 - Ficando preso a tecnologias ruins;
 - Entendendo mal;
 - Com projeto ruim ou nenhum projeto;
 - Reinventando a roda;
 - Violando o propósito de seu software.
- *Regra:* Você pode determinar se uma tecnologia é “ruim” ou não ao observar seu potencial de sobrevivência, interoperabilidade e atenção à qualidade.
- *Regra:* Com frequência, se algo está ficando muito complexo, isso significa que há um erro no projeto em algum lugar abaixo do nível onde a complexidade aparece.
- *Regra:* Quando se deparar com complexidade, pergunte: “Qual problema você está tentando resolver?”
- *Regra:* Os problemas mais difíceis de projeto podem ser resolvidos simplesmente desenhando ou anotando-os em papel.
- *Regra:* Para lidar com a complexidade em seu sistema, reprojete as partes individuais em pequenos passos.
- *Fato:* A pergunta-chave por trás de todas as simplificações válidas é: “Como isso poderia ser mais fácil de se lidar ou mais compreensível?”

- *Regra:* Se você se deparar com uma complexidade incorrigível fora de seu programa, coloque um envoltório (wrapper) em torno dela que seja simples para outros programadores.
- *Regra:* Reescrever é aceitável apenas em um conjunto muito limitado de situações.
- *Lei:* A Lei de testes: O grau em que você sabe como seu software se comporta é o grau em que você o testou com precisão.
- *Regra:* A menos que tenha tentado, você não sabe que ele funciona.