



Série de Robert C. Martin

Código Limpo

Habilidades Práticas do Agile Software

Prefácio

Um de nossos doces favoritos aqui na Dinamarca é o Ga-Jol, cujos fortes vapores de licorice são um complemento perfeito para nosso clima úmido e, geralmente, frio. Parte do charme do Ga-Jol, para nós dinamarqueses, são os dizeres sábios impressos em cada caixa. Comprei dois pacotes dessa iguaria essa manhã e nela veio este antigo ditado dinamarquês:

Ærlighed i små ting er ikke nogen lille ting.

“Honestidade em pequenas coisas não é uma coisa pequena”. Era um bom presságio para com o que eu já desejava dizer aqui. Pequenas coisas são importantes. Este é um livro sobre preocupações modestas cujos valores estão longe de ser pequenos.

Deus está nos detalhes, disse o arquiteto Ludwig Mies van der Rohe. Essa citação retoma argumentos com contemporâneos sobre o papel da arquitetura no desenvolvimento de software, especialmente no mundo Agile. Bob e eu, às vezes, acabávamos engajados entusiasmadamente debatendo sobre este assunto. E sim, Mies van der Rohe atentava para os utilitários e as formas imemoriais de construção que fundamentam uma ótima arquitetura. Por outro lado, ele também selecionava pessoalmente cada maçaneta para cada casa que ele projetava. Por quê? Por que pequenas coisas são importantes.

Em nosso “debate” sobre Desenvolvimento dirigido a testes (TDD, sigla em inglês), Bob e eu descobrimos que concordamos que a arquitetura do software possuir um lugar importante no desenvolvimento, embora provavelmente tenhamos perspectivas diferentes do significado exato disso. Essas diferenças são relativamente irrelevantes contudo, pois podemos admitir que profissionais responsáveis dedicam algum tempo para pensar e planejar o início de um projeto. As noções de desenvolvimento dirigido apenas por testes e por códigos do final da década de 1990 já não existem mais. Mesmo assim, a atenção aos detalhes é um fundamento de profissionalismo ainda mais crítico do que qualquer visão maior. Primeiro, é por meio da prática em pequenos trabalhos que profissionais adquirem proficiência e confiança para se aventurar nos maiores. Segundo, a menor parte de uma construção desleixada, a porta que não fecha direito ou o azulejo levemente torto do chão, ou mesmo uma mesa desarrumada, retiram completamente o charme do todo. É sobre isso que se trata o código limpo.

Ainda assim, a arquitetura é apenas uma metáfora para o desenvolvimento de software, especialmente para a parte que entrega o produto inicial no mesmo sentido que um arquiteto entrega uma construção imaculada. Nessa época do Scrum e do Agile, o foco está em colocar o

produto rapidamente no mercado. Desejamos que a indústria funcione em velocidade máxima na produção de software. Essas fábricas humanas: programadores que pensam e sentem que trabalham a partir das pendências de um produto ou do user story para criar o produto. A metáfora da fabricação está mais forte do que nunca no pensamento. Os aspectos da produção da manufatura japonesa de automóveis, de um mundo voltado para a linha de montagem, inspiraram grande parte do Scrum.

Ainda assim, mesmo na indústria automobilística, a maior parte do trabalho não está na fabricação, mas na manutenção – ou na prevenção. Em software, 80% ou mais do que fazemos é chamado de “manutenção”: o ato de reparar. Em vez de abraçar o típico foco ocidental sobre a produção de bons softwares, deveríamos pensar mais como um pedreiro que conserta casas na indústria de construção, ou um mecânico de automóveis na área automotiva. O que o gerenciamento japonês tem a dizer sobre isso?

Por volta de 1951, uma abordagem qualitativa chamada Manutenção Produtiva Total (TPM) surgiu no cenário japonês. Seu foco era na manutenção em vez da produção. Um dos maiores fundamentos da TPM é o conjunto dos chamados 5S princípios. 5S é uma série de disciplinas – uso aqui o termo “disciplina” para fins educativos. Os 5S princípios, na verdade, são os fundamentos do Lean – outro jargão no cenário ocidental, e cada vez mais conhecida no mundo dos softwares. Esses princípios não são uma opção. Assim como Uncle Bob diz em suas preliminares, a prática de um bom software requer tal disciplina: foco, presença de espírito e pensamento. Nem sempre é sobre fazer, sobre pressionar os equipamentos da fábrica para produzir em velocidade máxima. A filosofia dos 5S inclui os seguintes conceitos:

Seiri, ou organização (pense em “ordenar”). Saber onde estão as coisas – usar abordagens como nomes adequados – é crucial. Acha que dar nome a identificadores não é importante? Leia próximos capítulos.

Seiton, ou arrumação (pense em “sistematizar”). Há um antigo ditado americano que diz: “Um lugar para tudo, e tudo em seu lugar”. Um pedaço de código deve estar onde você espera encontrá-lo – caso não esteja, refatore e o coloque lá.

Seiso, ou limpeza (pensem em “polir”): manter o local de trabalho livre de fios pendurados, gordura, migalhas e lixo. O que os autores falam aqui sobre encher seu código com comentários e linhas de códigos como comentários que informa o passado ou os desejos para o futuro? Livre-se deles.

Seiketsu, ou padronização: a equipe concorda em manter o local de trabalho limpo.

Você acha que este livro fala algo sobre ter um estilo de programação consistente e uma série de práticas dentro da equipe? De onde vêm tais padrões? Continue a leitura.

Shitsuke, ou disciplina (autodisciplina). Isso significa ter disciplina para seguir as práticas e refletir frequentemente isso no trabalho e estar disposto a mudar.

Se aceitar o desafio – isso, o desafio – de ler e aplicar o que é aconselhado neste livro, você entenderá e apreciará o último item. Aqui estamos finalmente indo em direção às raízes do profissionalismo responsável numa profissão que deve se preocupar com o ciclo de vida de um produto. Conforme façamos a manutenção de automóveis e outras máquinas na TPM, a manutenção corretiva – esperar que bugs apareçam – é a exceção. Em vez disso, subimos um nível: inspecionamos as máquinas todos os dias e consertamos as partes desgastadas antes de quebrarem, ou percorremos o equivalente aos famosos 16km antes da primeira troca de óleo para testar o desgaste. No código, a refatoração é impiedosa. Você ainda pode melhorar um nível a mais com o advento do movimento da TPM há 50 anos; construa máquinas que sejam passíveis de manutenção. Tornar seu código legível é tão importante quanto torná-lo executável. A última prática, adicionada à TPM em torno de 1960, é focar na inclusão de máquinas inteiramente novas ou substituir as antigas. Como nos adverte Fred Brooks, provavelmente devemos refazer partes

principais do software a partir do zero a cada sete anos ou então se livrar dos entulhos. Talvez devêssemos atualizar a constante de tempo de Fred para semanas, dias ou horas, em vez de anos. É aí onde ficam os detalhes.

Há um grande poder nos detalhes, mesmo assim existe algo simples e profundo sobre essa abordagem para a vida, como talvez esperemos de qualquer abordagem que afirme ter origem japonesa. Mas essa não é apenas uma visão ocidental de mundo sobre a vida; a sabedoria de ingleses e americanos também está cheia dessas advertências. A citação de Seiton mais acima veio da ponta da caneta de um ministro em Ohio que visualizou literalmente a organização “como um remédio para todos os níveis de mal”.

E Seiso? Deus ama a limpeza. Por mais bonita que uma casa seja, uma mesa desarrumada retira seu esplendor. E Shutsuke nessas pequenas questões? Quem é fiel no pouco também é no muito. E que tal ficar ansioso para refatorar na hora certa, fortalecendo sua posição para as “grandes” decisões subsequentes, em vez de descartá-las? Um homem prevenido vale por dois. Deus ajuda a quem cedo madruga. Não deixe para amanhã o que se pode fazer hoje. (Esse era o sentido original da frase “o último momento de responsabilidade”, em Lean, até cair nas mãos dos consultores de software). E que tal aplicar o local de esforços pequenos e individuais num grande todo? De grão em grão a galinha enche o papo. Ou que tal integrar um trabalho de prevenção no dia a dia? Antes prevenir do que remediar. O código limpo honra as profundas raízes do conhecimento sob nossa cultura mais ampla, ou como ela fora um dia, ou deve ser, e poderá vir a ser com a atenção correta aos detalhes.

Mesmo na grande literatura na área de arquitetura encontramos visões que remetem a esses supostos detalhes. Pense nas maçanetas de Mies van der Rohe. Aquilo é seiri. É ficar atento a cada nome de variável. Deve-se escolher o nome de uma variável com cuidado, como se fosse eu primeiro filho.

Como todo proprietário de uma casa sabe, tal cuidado e constante refinamento jamais acaba.

O arquiteto Christopher Alexander – pai dos padrões e das linguagens de padrões – enxerga cada o próprio design como um conserto pequeno e local. E ele enxerga a habilidade de uma boa estrutura como o único objetivo do arquiteto; pode-se deixar as formas maiores para os padrões e seus aplicativos para os moradores. O design é constante não só ao adicionarmos um novo cômodo a uma casa, mas ao nos atentarmos a repintura, a substituição de carpetes gastos ou a melhoria da pia da cozinha. A maioria das artes reflete relações análogas. Em nossa busca por outras pessoas que dizem que a casa de Deus foi feita nos mínimos detalhes, encontramo-nos em boa companhia do autor francês Gustav Flaubert, do século XIX. O poeta francês Paul Valery nos informa que um poema nunca fica pronto e requer trabalho contínuo, e parar de trabalhar nele seria abandoná-lo.

Tal preocupação com os detalhes é comum a todos os encargos de excelência. Portanto, talvez haja pouca coisa nova aqui, mas ao ler este livro você será desafiado a retomar a disciplina que você há muito largou para a apatia ou um desejo pela espontaneidade e apenas “respondia às mudanças”.

Infelizmente, não costumamos enxergar essas questões como peças fundamentais da arte de programar. Abandonamos nosso código antecipadamente, não porque ele já esteja pronto, mas porque nosso sistema de valores se foca mais na aparência externa do que no conteúdo que entregamos.

No final, essa falta de atenção nos custa: Dinheiro ruim sempre reaparece. Pesquisas, nem no mercado e nem nas universidades, são humildes o bastante para se rebaixar e manter o código limpo. Na época em que trabalhei na empresa Bell Labs Software Production Research (produção, de fato!), ao ficarmos mexendo aqui e ali, nos deparamos com descobertas que sugeriam que o

estilo consistente de endentação era um dos indicadores mais significantes estatisticamente da baixa incidência de bugs.

Queríamos que a qualidade fosse produzida por essa ou aquela estrutura ou linguagem de programação ou outra noção de alto nível; conforme as pessoas cujo suposto profissionalismo se dá ao domínio de ferramentas e métodos de design grandioso, sentimo-nos ofendidos pelo valor que aquelas máquinas de fábricas, os codificadores, recebem devido a simples aplicação consistente em um estilo de endentação. Para citar meu próprio livro de 17 anos atrás, tal estilo faz a distinção entre excelência e mera competência. A visão de mundo japonesa entende o valor crucial do trabalhador diário e, além do mais, dos sistemas de desenvolvimento voltados para as ações simples e diárias desses trabalhadores. A qualidade é o resultado de um milhão de atos altruístas de importar-se – não apenas um grande método qualquer que desça dos céus. Não é porque esses atos são simples que eles sejam simplistas, e muito menos que sejam fáceis. Eles são, não obstante, a fábrica de magnitude e, também, de beleza em qualquer esforço humano. Ignorá-los é não ser ainda completamente humano.

É claro que ainda defendo o conceito de um escopo mais amplo, e, especialmente, o do valor de abordagens arquitetônicas arraigadas profundamente no conhecimento do domínio e de usabilidade do software.

Este livro não é sobre isso – ou, pelo menos, não de modo direto. Mas ele passa uma mensagem mais sutil cuja essência não deve ser subestimada. Ele se encaixa à máxima atual das pessoas que realmente se preocupam com o código, como Peter Sommerlad, Kevlin Henney e Giovanni. “O código é o projeto” e “Código simples” são seus mantras. Enquanto devamos tentar nos lembrar de que a interface é o programa, e que suas estruturas dizem bastante sobre a estrutura de nosso programa, é crucial adotar a humilde postura de que o projeto vive no código. E enquanto o retrabalho na metáfora da manufatura leva ao custo, o no projeto leva ao valor. Devemos ver nosso código como a bela articulação dos nobres esforços do projeto – projeto como um processo, e não uma meta estática. É no código que ocorrem as medidas estruturais de acoplamento e coesão. Se você vir a descrição de Larry Constantine sobre esses dois fatores, ele os conceitua em termos de código – e não em conceitos de alto nível como se pode encontrar em UML. Richard Gabriel nos informa em seu artigo *Abstraction Descant* que a abstração é maligna. O código é antimaligno, e talvez o código limpo seja divino.

Voltando à minha pequena embalagem de Ga-Jol, acho importante notar que a sabedoria dinamarquesa nos aconselha a não só prestar atenção a pequenas coisas, mas também a ser honesto em pequenas coisas. Isso significa ser honesto com o código e tanto com nossos colegas e, acima de tudo, com nós mesmos sobre o estado de nosso código. Fizemos o Melhor para “deixar o local mais limpo do que como o encontramos”? Refatoramos nosso código antes de verificar-lo? Essas não são preocupações externas, mas preocupações que estão no centro dos valores do Agile. Que a refatoração seja parte do conceito de “Pronto”, é uma prática recomendada no Scrum. Nem a arquitetura e nem o código limpo exigem perfeição, apenas honestidade e que façamos o melhor de nós. Errar é humano; perdoar é divino. No Scrum, tornamos as coisas visíveis. Arejamos nossa roupa suja. Somos honestos sobre o estado de nosso código porque o código nunca é perfeito. Tornamo-nos mais completamente humanos, mais merecedores do divino e mais próximos da magnitude dos detalhes.

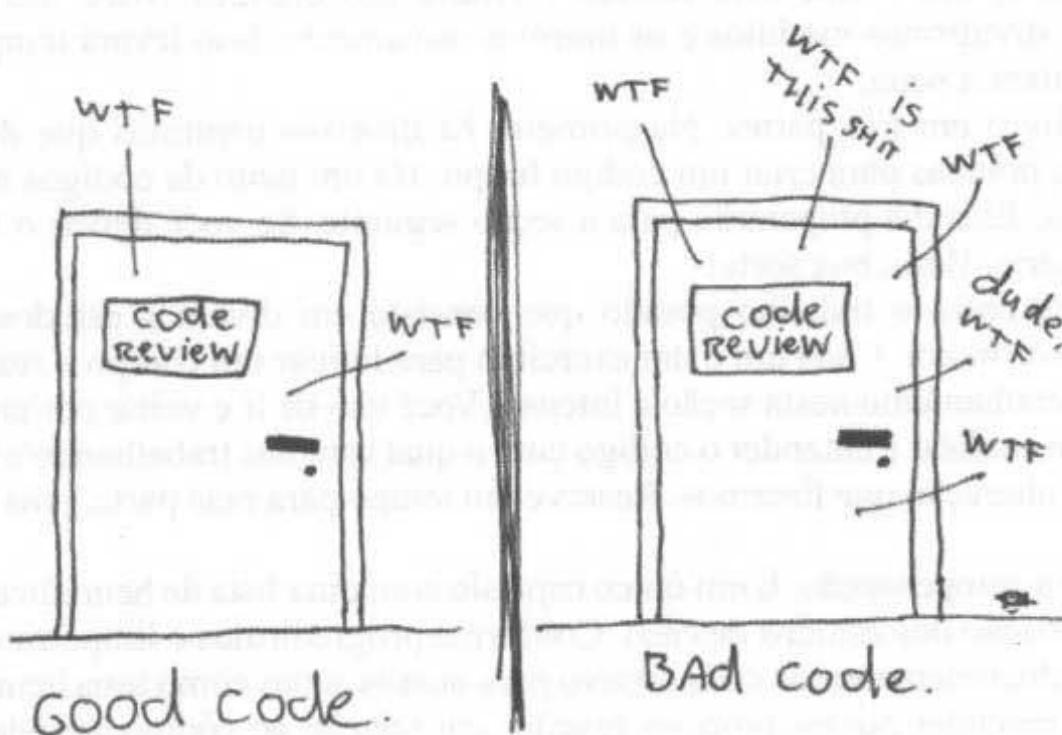
Em nossa profissão, precisamos desesperadamente de toda ajuda que conseguirmos. Se o piso de uma loja reduz os acidentes e suas ferramentas bem organizadas aumentam a produtividade, então sou totalmente a favor. E relação a este livro, ele é a melhor aplicação pragmática dos princípios de Lean ao software que já vi. Eu não esperava nada mais deste pequeno grupo prático de indivíduos que se esforçaram juntos por anos não só para se aperfeiçoarem, mas também para presentear com seus conhecimentos o mercado com obras como esta em suas mãos agora.

Isso deixa o mundo um pouco melhor do que quando o encontrei antes de Uncle Bob me enviar o manuscrito. Após ter finalizado estes exercícios com conhecimentos tão sublimes, agora vou limpar minha mesa.

James O. Coplien
Mørdrup, Dinamarca

Introdução

The ONLY VALID MEASUREMENT
OF Code QUALITY: WTFs/MINUTE



Reproduzido com a gentil autorização de Thom Holwerda.

http://www.osnews.com/story/19266/WTFs_m

(c) 2008 Focus Shift

Que porta representa seu código? Que porta representa sua equipe ou sua companhia?

Por que estamos naquela sala? É apenas uma revisão normal de código ou encontramos uma série de problemas terríveis logo após iniciarmos a apresentação? Estamos depurando em pânico, lendo meticulosamente um código que pensávamos que funcionava? Os clientes estão indo embora aos bando e estamos com os gerentes em nossos pescoços? Como podemos garantir que chegaremos atrás da porta certa quando o caminho fica difícil? A resposta é: habilidade profissional.

Há duas vertentes para se obter habilidade profissional: conhecimento e trabalho. Você deve adquirir o conhecimento dos princípios, padrões, práticas e heurísticas que um profissional

habilidoso sabe, e também esmiuçar esse conhecimento com seus dedos, olhos e corpo por meio do trabalho árduo e da prática.

Posso lhe ensinar a mecânica para se andar de bicicleta. Na verdade, a matemática clássica é relativamente direta. Gravidade, atrito, momento angular, centro de massa, e assim por diante, podem ser demonstrados com menos de uma página cheia de equações. Dada essas fórmulas, eu poderia provar para você que é prático andar de bicicleta e lhe dar todo o conhecimento necessário para que você consiga. E mesmo assim você cairá na primeira vez que tentar.

Programar não é diferente. Poderíamos pôr no papel todos os princípios necessários para um código limpo e, então, confiar que você fará as tarefas (isto é, deixar você cair quando subir na bicicleta), mas que tipo de professores e de estudantes isso faria de nós?

Não. Essa não é a forma que este livro seguirá.

Aprender a criar códigos limpos é uma tarefa árdua e requer mais do que o simples conhecimento dos princípios e padrões. Você deve suar a camisa; praticar sozinho e ver que cometeu erros; assistir a outros praticarem e errarem; vê-los tropeçar e refazer seus passos; Vê-los agonizar para tomar decisões e o preço que pagarão por as terem tomado da maneira errada.

Esteja preparado para trabalhar duro enquanto ler este livro. Esse não é um livro fácil e simples que você pode ler num avião e terminar antes de aterrissar. Este livro lhe fará trabalhar, e trabalhar duro. Que tipo de trabalho você fará? Você lerá códigos aqui, muitos códigos.

E você deverá descobrir o que está correto e errado nos códigos. Você terá de seguir o raciocínio conforme dividirmos módulos e os unirmos novamente. Isso levará tempo e esforço, mas achamos que valerá a pena.

Dividimos este livro em três partes. Na primeira há diversos capítulos que descrevem os princípios, padrões e práticas para criar um código limpo. Há um tanto de códigos nessa parte, e será desafiador lê-los. Eles lhe prepararão para a seção seguinte. Se você deixar o livro de lado após essa primeira parte. Bem, boa sorte!

Na segunda parte entra o trabalho pesado que consiste em diversos estudos de caso de complexidade cada vez maior. Cada um é um exercício para limpar um código – resolver alguns problemas dele. O detalhamento nesta seção é intenso. Você terá de ir e voltar por entre as folhas de textos e códigos, e analisar e entender o código com o qual estamos trabalhando e captar nosso raciocínio para cada alteração que fizermos. Reserve um tempo para essa parte, pois deverá levar dias.

A terceira parte é a compensação. É um único capítulo com uma lista de heurísticas e “odores” reunidos durante a criação dos estudos de caso. Conforme progredirmos e limparmos os códigos nos estudos de caso, documentaremos cada motivo para nossas ações como uma heurística ou um “odor”. Tentaremos entender nossas próprias reações em relação ao código quando estivermos lendo ou alterando-o, e trabalharemos duro para captar por que nos sentimos de tal forma e por que fizemos isso ou aquilo. O resultado será um conhecimento base que descreve a forma como pensamos quando criamos, lemos e limpamos um código.

Este conhecimento base possui um valor limitado se você não ler com atenção ao longo dos casos de estudo na segunda parte deste livro. Neles, anotamos cuidadosamente cada alteração que fizemos com referências posteriores às heurísticas. Tais referências aparecem em colchetes, assim: [H22]. Isso lhe permite ver o contexto no qual são aplicadas e escritas aquelas heurísticas! Essas, em si, não são tão valiosas, mas, sim, a relação entre elas e as diferentes decisões que tomamos ao limpar o código nos casos de estudo.

A fim de lhe ajudar ainda mais com essas relações, colocamos no final do livro uma referência cruzada que mostra o número da página para cada referência. Você pode usá-la com um guia rápido de consulta para saber onde uma determinada heurística foi aplicada.

Mesmo se você ler a primeira e a terceira seções e pular para os casos de estudo, ainda

assim terá lido um livro que satisfaz sobre a criação de um bom software. Mas se não tiver pressa e explorar os casos de estudo, seguindo cada pequeno passo e cada minuto de decisão, se colocando em nosso lugar e se forçando a seguir a mesma linha de raciocínio que usamos, então você adquiriu um entendimento muito mais rico de todos aqueles princípios, padrões, práticas e heurísticas. Todo esse conhecimento ficará em cada fibra de seu corpo. Ele se tornará parte de você da mesma forma que ao aprender a andar de bicicleta, ela se torna uma parte de sua vontade de pedalá-la.

Agradecimentos

Ilustrações

Meus agradecimentos a minhas duas artistas, Jeniffer Kohnke e Angela Brooks. Jennifer é a responsável pelas maravilhosas e criativas figuras no inicio de cada capítulo e também pelos retratos de Kent Beck, Ward Cunningham, Bjarne Stroustrup, Ron Jeffries, Grady Booch, Dave Thomas, Michael Feathers e de mim mesmo.

Angela é a responsável pelas figuras engenhosas que enfeitam os capítulos. Ao longo dos anos, ela criou bastantes imagens para mim, incluindo muitas das que estão no livro Agile Software Development: Principles, Patterns, and Practices. Angela também é minha primogênita e de quem me sinto muito orgulhoso.

Sobre a capa

A imagem da capa se é a M104 – a Galáxia Sombrero –, que fica na constelação de Virgem e está a pouco menos de 30 milhões de anos-luz de nós. Em seu centro há um buraco negro supermassivo cujo peso equivale um bilhão de vezes a massa do sol.

A imagem lhe faz lembrar da explosão da lua Praxis, de Klingon? Eu me recordo vividamente a cena de Jornada nas Estrelas VI que mostrava um anel equatorial de destroços voando devido à explosão. Desde essa cena, o anel equatorial se tornou um componente comum às explosões em filmes de ficção científica. Ele até foi adicionado à explosão de Alderaan nas últimas versões do filme Jornada nas Estrelas.

O que causou a formação desse anel em torno de M104? Por que ele possui um bojo tão amplo e um núcleo tão minúsculo e brilhoso? Parece-me como se o buraco negro central perdeu sua graça e lançou um buraco de 30.000 anos-luz no meio da galáxia, devastando quaisquer civilizações que estivessem no caminho daquele distúrbio cósmico.

Buracos negros supermassivos engolem estrelas inteiras no almoço, convertendo uma considerável fração de sua massa em energia. $E = MC^2$ já é bastante potência, mas quando M é uma massa estelar:

Cuidado! Quantas estrelas caíram impetuosamente naquelas presas antes de o monstro ficar saciado?

O tamanho do vão central poderia ser uma dica?

A imagem de M104 da capa é uma combinação da famosa fotografia de luz visível do Hubble (foto superior) com a recente imagem infravermelha do observatório espacial Spitzer (foto inferior). É essa segunda imagem que nos mostra claramente a natureza do anel da galáxia. Na luz visível, vemos apenas a extremidade frontal do anel como uma silhueta. O bojo central ofusca o resto do anel.

Mas no infravermelho, as partículas "quentes" – isto é, altamente radioativas – no anel brilham através do bojo central. A combinação de ambas as imagens nos mostra uma visão que não havíamos visto antes e indica que, há muito tempo atrás, lá havia um inferno enfurecido de atividades.



Código Limpo



Há duas razões pelas quais você está lendo este livro: você é programador e deseja se tornar um ainda melhor. Ótimo. Precisamos de programadores melhores.

Este livro fala sobre programação e está repleto de códigos que examinaremos a partir de diferentes perspectivas: de baixo para cima, de cima para baixo e de dentro para fora. Ao terminarmos, teremos um amplo conhecimento sobre códigos e seremos capazes de distinguir entre um código bom e um código ruim. Saberemos como escrever um bom código e como tornar um ruim em um bom.

O Código

Podem dizer que um livro sobre códigos é, de certa forma, algo ultrapassado, que a programação deixou de ser uma preocupação e que devemos nos preocupar com modelos e requisitos. Outros até mesmo alegam que o fim do código, ou seja, da programação, está próximo; que logo todo código será gerado, e não mais escrito. E que não precisarão mais de programadores, pois as pessoas criarião programas a partir de especificações.

Bobagens! Nunca nos livraremos dos códigos, pois eles representam os detalhes dos requisitos. Em certo nível, não há como ignorar ou abstrair esses detalhes; eles precisam ser especificados. E especificar requisitos detalhadamente de modo que uma máquina possa executá-los é *programar* – e tal especificação é o código.

Espero que o nível de de nossas linguagens continue a aumentar e que o número de linguagens específicas a um domínio continue crescendo. Isso será bom, mas não acabará com a programação. De fato, todas as especificações escritas nessas linguagens de níveis mais altos e específicas a um domínio *serão códigos!* Eles precisarão ser minuciosos, exatos e bastante formais e detalhados para que uma máquina possa entendê-los e executá-los.

As pessoas que pensam que o código um dia desaparecerá são como matemáticos que esperam algum dia descobrir uma matemática que não precise ser formal. Elas esperam que um dia descubramos uma forma de criar máquinas que possam fazer o que desejamos em vez do que mandamos. Tais máquinas terão de ser capazes de nos entender tão bem de modo que possam traduzir exigências vagamente especificadas em programas executáveis perfeitos para satisfazer nossas necessidades.

Isso jamais acontecerá. Nem mesmo os seres humanos, com toda sua intuição e criatividade, têm sido capazes de criar sistemas bem-sucedidos a partir das carências confusas de seus clientes. Na verdade, se a matéria sobre especificação de requisitos não nos ensinou nada, é porque os requisitos bem especificados são tão formais quanto os códigos e podem agir como testes executáveis de tais códigos!

Lembre-se de que o código é a linguagem na qual expressamos nossos requisitos. Podemos criar linguagens que sejam mais próximas a eles. Podemos criar ferramentas que nos ajudem a analisar a sintaxe e unir tais requisitos em estruturas formais. Mas jamais eliminaremos a precisão necessária – portanto, sempre haverá um código.

Código ruim

Recentemente li o prefácio do livro *Implementation Patterns*¹ de Kent Beck, no qual ele diz "... este livro baseia-se numa premissa frágil de que um bom código importa...". Uma premissa *frágil*? Não concordo! Acho que essa premissa é uma das mais robustas, apoiadas e plenas do que todas as outras em nossa área (e sei que Kent sabe disso). Estamos cientes de que um bom código importa, pois tivemos de lidar com a falta dele por muito tempo.

Lembro que no final da década de 1980 uma empresa criou um aplicativo extraordinário que se tornou muito popular e muitos profissionais o compraram e usaram. Mas, então, o intervalo entre os lançamentos das novas distribuições começou a aumentar. Os *bugs* não eram consertados na distribuição seguinte. E o tempo de carregamento do aplicativo e o número de travamentos aumentaram. Lembro-me do dia em que, frustrado, fechei o programa e nunca mais o usei. A empresa saiu do mercado logo depois.

Duas décadas depois encontrei um dos funcionários de tal empresa na época e o perguntei o que havia acontecido, e o que eu temia fora confirmado. Eles tiveram de apressar o lançamento do produto e, devido a isso, o código ficou uma zona. Então, conforme foram adicionando mais e mais recursos, o código piorava cada vez mais até que simplesmente não era mais possível gerenciá-lo. *Foi o código ruim que acabou com a empresa.*

Alguma vez um código ruim já lhe atrasou consideravelmente? Se você for um programador, independente de sua experiência, então já se deparou várias vezes com esse obstáculo. Aliás, é como se caminhássemos penosamente por um lamaçal de arbustos emaranhados com armadilhas ocultas. Isso é o que fazemos num código ruim. Pelejamos para encontrar nosso caminho, esperando avistar alguma dica, alguma indicação do que está acontecendo; mas tudo o que vemos é um código cada vez mais sem sentido.

É claro que um código ruim já lhe atrasou. Mas, então, por que você o escreveu dessa forma? Estava tentando ser rápido? Estava com pressa? Provavelmente. Talvez você pensou que não tivesse tempo para fazer um bom trabalho; que seu chefe ficaria com raiva se você demorasse um pouco mais para limpar seu código. Talvez você estava apenas cansado de trabalhar neste programa e queria terminá-lo logo. Ou verificou a lista de coisas que havia prometido fazer e percebeu que precisava finalizar este módulo de uma vez, de modo que pudesse passar para o próximo. Todos já fizemos isso, já vimos a bagunça que fizemos e, então, optamos por arrumá-las outro dia. Todos já nos sentimos aliviados ao vermos nosso programa confuso funcionar e decidimos que uma bagunça que funciona é melhor do que nada. Todos nós já dissemos que revisaríamos e limparíamos o código depois. É claro que naquela época não conhecíamos a lei de LeBlanc: *Nunca é tarde.*



O Custo de Ter um Código Confuso

Se você é programador a mais de dois ou três anos, provavelmente o código confuso de outra pessoa já fez com que você trabalhasse mais lentamente e provavelmente seu próprio código já lhe trouxe problemas.

O nível de retardo pode ser significativo. Ao longo de um ou dois anos, as equipes que trabalharam rapidamente no início de um projeto podem perceber mais tarde que estão indo a passos de tartaruga. Cada alteração feita no código causa uma falha em outras duas ou três partes do mesmo código. Mudança alguma é trivial. Cada adição ou modificação ao sistema exige que restaurações, amarrações e remendos sejam “entendidas” de modo que outras possam ser incluídas. Com o tempo, a bagunça se torna tão grande e profunda que não dá para arrumá-la. Não há absolutamente solução alguma.

Conforme a confusão aumenta, a produtividade da equipe diminui, assintoticamente aproximando-se de zero. Com a redução da produtividade, a gerência faz a única coisa que ela pode; adiciona mais membros ao projeto na esperança de aumentar a produtividade. Mas esses novos membros não conhecem o projeto do sistema, não sabem a diferença entre uma mudança que altera o propósito do projeto e aquela que o atrapalha. Ademais, eles, e todo o resto da equipe, estão sobre tremenda pressão para aumentar a produtividade. Com isso todos criam mais e mais confusões, levando a produtividade mais perto ainda de zero (veja a Figura 1.1).

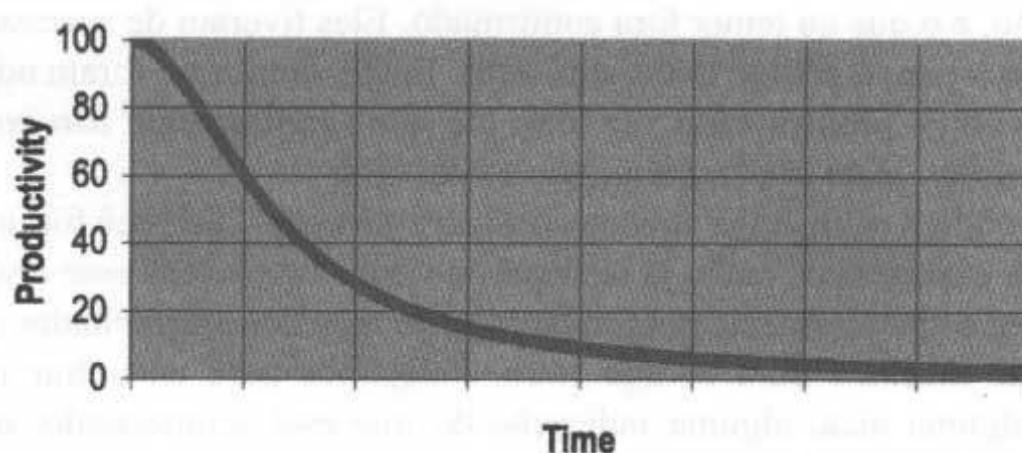


Figura 1.1
Produtividade v. tempo

O Grande Replanejamento

No final, a equipe se rebela. Todos informam à gerência que não conseguem mais trabalhar neste irritante código-fonte e exigem um replanejamento do projeto. Apesar de a gerência não querer gastar recursos em uma nova remodelação, ela não pode negar que a produtividade está péssima. No final das contas, ela acaba cedendo às exigências dos desenvolvedores e autoriza o grande replanejamento desejado.

É, então, formada uma nova equipe especializada. Por ser um projeto inteiramente novo, todos querem fazer parte dessa equipe. Eles desejam começar do zero e criar algo belo de verdade. Mas apenas os melhores e mais brilhantes são selecionados e os outros deverão continuar na manutenção do sistema atual.

Agora ambos os times estão numa corrida. A nova equipe precisa construir um novo sistema que faça o mesmo que o antigo, além de ter de se manter atualizada em relação às mudanças feitas constantemente no sistema antigo. Este, a gerência não substituirá até que o novo possa fazer tudo também.

Essa corrida pode durar um bom tempo. Já vi umas levarem 10 anos. E, quando ela termina, os membros originais da nova equipe já foram embora há muito tempo, e os atuais exigem o replanejamento de um novo sistema, pois está tudo uma zona novamente.

Se você já vivenciou pelo menos um pouco dessa situação, então sabe que dedicar tempo para limpar seu código não é apenas eficaz em termos de custo, mas uma questão de sobrevivência profissional.

Atitude

Você já teve de trabalhar penosamente por uma confusão tão grave que levou semanas o que deveria ter levado horas? Você já presenciou o que deveria ser uma alteração única e direta, mas que em vez disso foi feita em diversos módulos distintos? Todos esses sintomas são bastante comuns.

Por que isso ocorre em um código? Por que um código bom se decompõe tão rápido em um ruim? Temos diversas explicações para isso. Reclamamos que os requisitos mudaram de tal forma que estragaram o projeto original. Criticamos os prazos por serem curtos demais para fazermos as coisas certas. Resmungamos sobre gerentes tolos e clientes intolerantes e tipos de marketing inúteis e técnicos de telefone. Mas o padrão, querido Dilbert¹, não está em nossas estrelas, mas sim em nós mesmos. Somos profissionais.

Isso pode ser algo difícil de engolir. Mas como poderia essa zona ser *nossa* culpa? E os requisitos? E o prazo? E os tolos gerentes e tipos de marketing inúteis? Eles não carregam alguma parcela da culpa?

Não. Os gerentes e marketeiros buscam em nós as informações que precisam para fazer promessas e firmarem compromissos; e mesmo quando não nos procuram, não devemos dar uma de tímidos ao dizer-lhes nossa opinião. Os usuários esperam que validemos as maneiras pelas quais os requisitos se encaixarão no sistema. Os gerentes esperam que os ajudemos a cumprir o prazo. Nossa cumplicidade no planejamento do projeto é tamanha que compartilhamos de uma grande parcela da responsabilidade em caso de falhas; especialmente se estas forem em relação a um código ruim.

“Mas, espere!”, você diz. “E se eu não fizer o que meu gerente quer, serei demitido”. É provável que não.

A maioria dos gerentes quer a verdade, mesmo que demonstrem o contrário. A maioria deles quer um código bom, mesmo estourando o prazo. Eles podem proteger com paixão o prazo e os requisitos, mas essa é a função deles. A *sua* é proteger o código com essa mesma paixão.

Para finalizar essa questão, e se você fosse médico e um paciente exigisse que você parasse com toda aquela lavação das mãos na preparação para a cirurgia só porque isso leva muito tempo?² É óbvio que o chefe neste caso é o paciente; mas, mesmo assim, o médico deverá totalmente se recusar obedecê-lo. Por quê? Porque o médico sabe mais do que o paciente sobre os riscos de doenças e infecções. Não seria profissional (sem mencionar criminoso) que o médico obedecesse ao paciente neste cenário. Da mesma forma que não é profissional que programadores cedam à vontade dos gerentes que não entendem os riscos de se gerar códigos confusos.

O Principal Dilema

Os programadores se deparam com um dilema de valores básicos. Todos os desenvolvedores com alguns anos a mais de experiência sabem que bagunças antigas reduzem o rendimento. Mesmo assim todos eles se sentem pressionados a cometer essas bagunças para cumprir os prazos. Resumindo, eles não se esforçam para.

Os profissionais sérios sabem que a segunda parte do dilema está errada. Você não cumprirá o prazo se fizer bagunça no código. De fato, tal desorganização reduzirá instantaneamente sua velocidade de trabalho, e você perderá o prazo. A *única* maneira de isso não acontecer – a única maneira de ir mais rápido – é sempre manter o código limpo.

A Arte do Código Limpo?

Digamos que você acredite que um código confuso seja um obstáculo relevante. Digamos que você aceite que a única forma de trabalhar mais rápido é manter seu código limpo. Então, você deve se perguntar: “Como escrever um código limpo?” Não vale de nada tentar escrever um código limpo se você não souber o que isso significa.

As más notícias são que escrever um código limpo é como pintar um quadro. A maioria de nós sabe quando a figura foi bem ou mal pintada. Mas ser capaz de distinguir uma boa arte de uma ruim não significa que você saiba pintar. Assim como saber distinguir um código limpo de um ruim não quer dizer que saibamos escrever um código limpo.

Escrever um código limpo exige o uso disciplinado de uma miríade de pequenas técnicas aplicadas por meio de uma sensibilidade meticulosamente adquirida sobre “limpeza”. A “sensibilidade ao código” é o segredo. Alguns de nós já nascemos com ela. Outros precisam se esforçar para adquiri-la. Ela não só nos permite perceber se o código é bom ou ruim, como também nos mostra a estratégia e disciplina de como transformar um código ruim em um limpo.

Um programador sem “sensibilidade ao código” pode visualizar um módulo confuso e reconhecer a bagunça, mas não saberá o que fazer a respeito dela. Já um com essa sensibilidade olhará um módulo confuso e verá alternativas. A “sensibilidade ao código” ajudará a esse

² Em 1847, quando Ignaz Semmelweis sugeriu pela primeira vez a lavagem das mãos, ela foi rejeitada, baseando-se no fato de que os médicos eram ocupados

programador a escolher a melhor alternativa e o orientará na criação de uma sequência de comportamentos para proteger as alterações feitas aqui e ali.

Em suma, um programador que escreve um código limpo é um artista que pode pegar uma tela em branco e submetê-la a uma série de transformações até que se torne um sistema graciosamente programado.

O que é um Código Limpo?

Provavelmente existem tantas definições como existem programadores. Portanto, perguntei a alguns programadores bem conhecidos e com muita experiência o que achavam.

Bjarne Stroustrup, criador do C++ e autor do livro *A linguagem de programação C++*

Gosto do meu código elegante e eficiente. A lógica deve ser direta para dificultar o encobrimento de bugs, as dependências mínimas para facilitar a manutenção, o tratamento de erro completo de acordo com uma estratégia clara e o desempenho próximo do mais eficiente de modo a não incitar as pessoas a tornarem o código confuso com otimizações sorrateiras. O código limpo faz bem apenas uma coisa.

Bjarne usa a palavra “elegante” – uma palavra e tanto! O dicionário possui as seguintes definições: *que se caracteriza pela naturalidade de harmonia, leveza, simplicidade; naturalidade no modo se dispor; requintado, fino, estiloso.* Observe a ênfase dada à palavra “naturalidade”.

Aparentemente, Bjarne acha que um código limpo proporciona uma leitura natural; e lê-lo deve ser belo como ouvir uma música num rádio ou visualizar um carro de design magnífico.

Bjarne também menciona duas vezes “eficiência”. Talvez isso não devesse nos surpreender vindo do criador do C++, mas acho que ele quis dizer mais do que um simples desejo por agilidade. A repetição de ciclos não é elegante, não é belo. E repare que Bjarne usa a palavra “incitar” para descrever a consequência de tal desleigânciia. A verdade aqui é que um código ruim incita o crescimento do caos num código. Quando outras pessoas alteram um código ruim, elas tendem a piorá-lo.

Pragmáticos, Dave Thomas e Andy Hunt expressam isso de outra forma. Eles usam a metáfora das janelas quebradas.³ Uma construção com janelas quebradas parece que ninguém cuida dela. Dessa forma, outras pessoas deixam de se preocupar com ela também. Elas permitem que as outras janelas se quebrem também. No final das contas, as próprias pessoas as quebram. Elas estragam a fachada com pichações e deixam acumular lixo. Uma única janela inicia o processo de degradação.



Bjarne também menciona que o tratamento de erro deva ser completo. Isso significa prestar atenção nos detalhes. Um tratamento de erro reduzido é apenas uma das maneiras pela qual os programadores deixam de notar os detalhes. Perdas de memória e condições de corrida são outras. Nomenclaturas inconsistentes são ainda outras. A conclusão é que um código limpo requer bastante atenção aos detalhes.

Bjarne conclui com a asseveração de que um código limpo faz bem apenas uma coisa. Não é por acaso que há inúmeros princípios de desenvolvimento de software que podem ser resumidos a essa simples afirmação. Vários escritores já tentaram passar essa ideia. Um código ruim tenta fazer coisas demais, ele está cheio de propósitos obscuros e ambíguos. O código limpo é *centralizado*. Cada função, cada classe, cada módulo expõe uma única tarefa que nunca sofre interferência de outros detalhes ou fica rodeada por eles.

Grady Booch, autor do livro *Object Oriented Analysis and Design with Applications*

Um código limpo é simples e direto. Ele é tão bem legível quanto uma prosa bem escrita. Ele jamais torna confuso o objetivo do desenvolvedor, em vez disso, ele está repleto de abstrações claras e linhas de controle objetivas.

Grady fala de alguns dos mesmos pontos que Bjarne, voltando-se mais para questão da legibilidade. Eu, particularmente, gosto desse ponto de vista de que ler um código limpo deve ser como ler uma prosa bem escrita.

Pense num livro muito bom que você já leu. Lembre-se de como as palavras eram substituídas por imagens! Era como assistir a um filme, não era? Melhor ainda, você via os personagens, ouvia os sons, envolvia-se nas emoções e no humor.

Ler um código limpo jamais será como ler *O senhor dos anéis*. Mesmo assim, a analogia com a literatura não é ruim. Como um bom romance, um código limpo deve expor claramente as questões do problema a ser solucionado. Ele deve desenvolvê-las até um clímax e, então, dar ao leitor aquele “Ahá! Mas é claro!”, como as questões e os suspenses que são solucionados na revelação de uma solução óbvia.

Acho que o uso que Grady faz da frase “abstrações claras” é um paradoxo fascinante! Apesar de tudo, a palavra “clara” é praticamente um sinônimo para “explicito”. Meu dicionário MacBook tem a seguinte definição para “claro(a)”: *direto, decisivo, sem devaneios ou detalhes desnecessários*. Apesar desta justaposição de significados, as palavras carregam uma mensagem poderosa. Nosso código deve ser decisivo, sem especulações. Ele deve conter apenas o necessário. Nossos leitores devem assumir que fomos decisivos.



O “grande” Dave Thomas, fundador da OTI, o pai da estratégia Eclipse

Além de seu criador, um desenvolvedor pode ler e melhorar um código limpo. Ele tem testes de unidade e de aceitação, nomes significativos; ele oferece apenas uma maneira, e não várias, de se fazer uma tarefa; possui poucas dependências, as quais são explicitamente declaradas e oferecem um API mínimo e claro. O código deve ser inteligível já que dependendo da linguagem, nem toda informação necessária pode expressa no código em si.

Dave compartilha do mesmo desejo de Grady pela legibilidade, mas com uma diferença relevante. Dave afirma que um código limpo facilita para que outras pessoas o melhorem. Pode parecer óbvio, mas não se deve enfatizar muito isso. Há, afinal de contas, uma diferença entre um código fácil de ler e um fácil de alterar.

Dave associa limpeza a testes! Dez anos atrás, isso levantaria um ar de desconfiança. Mas o estudo do Desenvolvimento Dirigido a Testes teve grande impacto em nossa indústria e se tornou uma de nossos campos de estudo mais essenciais. Dave está certo. Um código, sem testes, não está limpo. Não importa o quanto elegante, legível ou acessível esteja que, se ele não possuir testes, ele não é limpo.

Dave usa a palavra mínima duas vezes. Aparentemente ele dá preferência a um código pequeno. De fato, esse tem sido uma citação comum na literatura computacional. Quando menor, melhor. Dave também diz que o código deve ser *inteligível* – referência esta à *programação inteligível*⁴ (do livro *Literate Programming*) de Donald Knuth. A conclusão é que o código deve ser escrito de uma forma que seja inteligível aos seres humanos.

Michael Feathers, autor de *Working Effectively with Legacy Code*

Eu poderia listar todas as qualidades que vejo em um código limpo, mas há uma predominante que leva a todas as outras. Um código limpo sempre parece que foi escrito por alguém que se importava. Não há nada de óbvio no que se pode fazer para torná-lo melhor. Tudo foi pensado pelo autor do código, e se tentar pensar em algumas melhorias, você voltará ao início, ou seja, apreciando o código deixado para você por alguém que se importa bastante com essa tarefa.

One word: care. É esse o assunto deste livro. Talvez um subtítulo apropriado seria *Como se importar com o código*.

Michael bate na testa: um código limpo é um código que foi cuidado por alguém. Alguém que calmamente



o manteve simples e organizado; alguém que prestou a atenção necessária aos detalhes; alguém que se importou.

Ron Jeffries, autor de *Extreme Programming Installed and Extreme Programming Adventures in C#*

Ron iniciou sua carreira de programador em Fortran, no Strategic Air Command, e criou códigos em quase todas as linguagens e máquinas. Vale a pena considerar suas palavras:

Nestes anos recentes, comecei, e quase finalizei, com as regras de Beck sobre código simples. Em ordem de prioridade, são:

- Efetue todos os testes;
- Sem duplicação de código;
- Expressa todas as ideias do projeto que estão no sistema;
- Minimiza o número de entidades, como classes, métodos, funções e outras do tipo.

Dessas quatro, foco-me mais na de duplicação. Quando a mesma coisa é feita repetidas vezes, é sinal de que uma ideia em sua cabeça não está bem representada no código. Tento descobrir o que é e, então, expressar aquela ideia com mais clareza.

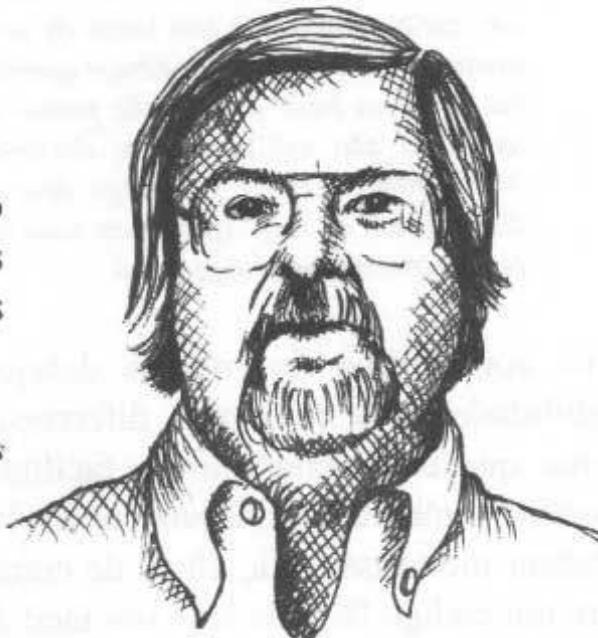
Expressividade para mim são nomes significativos e costume mudar o nome das coisas várias vezes antes de finalizar. Com ferramentas de programação modernas, como a Eclipse, renomear é bastante fácil, e por isso não me incomodo em fazer isso. Entretanto, a expressividade vai além de nomes. Também verifico se um método ou objeto faz mais de uma tarefa. Se for um objeto, provavelmente ele precisará ser dividido em dois ou mais. Se for um método, sempre uso a refatoração do Método de Extração nele, resultando em um método que expressa mais claramente sua função e em outros métodos que dizem como ela é feita.

Duplicação e expressividade me levam ao que considero um código limpo, e melhorar um código ruim com apenas esses dois conceitos na mente pode fazer uma grande diferença. Há, porém, uma outra coisa da qual estou ciente quando programo, que é um pouco mais difícil de explicar.

Após anos de trabalho, parece-me que todos os programadores pensam tudo igual. Um exemplo é “encontrar coisas numa coleção”. Tenhamos uma base de dados com registros de funcionários ou uma tabela hash de chaves e valores ou um vetor de itens de algum tipo, geralmente procuramos um item específico naquela coleção. Quando percebo isso, costumo implementar essa função em um método ou classe mais abstrato – o que me proporciona algumas vantagens interessantes.

Penso implementar a funcionalidade agora com algo mais simples, digamos uma tabela hash, mas como agora todas as referências àquela busca estão na minha classe ou método abstrato, posso alterar a implementação sempre que desejar. Posso ainda prosseguir rapidamente enquanto preservo a capacidade de alteração futura.

Além disso, a de coleções geralmente chama minha atenção para o que realmente está acontecendo, e impede que eu implemente funcionalidades arbitrárias em coleções quando tudo que eu preciso são simples maneiras de encontrar o que desejo. Redução de duplicação de código, alta expressividade e criação no início de abstrações simples. É isso que torna para mim um código limpo.



Aqui, em alguns breves parágrafos, Ron resumiu o conteúdo deste livro. Sem duplicação, uma tarefa, expressividade, pequenas abstrações. Tudo foi mencionado.

Ward Cunningham, criador do conceito de “WikiWiki”, criador do Fit, co-criador da Programação Extrema (eXtreme Programming). Incentivador dos Padrões de Projeto. Líder da Smalltalk e da OO. Pai de todos aqueles que se importam com o código.

Você sabe que está criando um código limpo quando cada rotina que você leia se mostra como o que você esperava. Você pode chamar de código belo quando ele também faz parecer que a linguagem foi feita para o problema.



Declarações como essa são características de Ward. Você a lê, coloca na sua cabeça e segue para a próxima. Parece tão racional, tão óbvio que raramente é memorizada como algo profundo. Você acha que basicamente já pensava assim. Mas observemos mais atentamente.

“... o que você esperava”. Qual foi a última vez que você viu um módulo como você o esperava que fosse? Não é mais comum eles serem complexos, complicados, emaranhados? Interpretá-lo erroneamente não é a regra? Você não está acostumado a se descontrolar ao tentar entender o raciocínio que gerou todo o sistema e associá-lo ao módulo que estás lendo? Quando foi a última vez que você leu um código para o qual você assentiu com a cabeça da mesma forma que fez com a declaração de Ward?

Este espera que ao ler um código limpo nada lhe surpreenda. De fato, não será nem preciso muito esforço. Você irá lê-lo e será basicamente o que você já esperava. O código é óbvio, simples e convincente. Cada módulo prepara o terreno para o seguinte. Cada um lhe diz como o próximo estará escrito. Os programas que são tão limpos e claros assim foram tão bem escritos que você nem perceberá. O programador o faz parecer super simples, como o é todo projeto extraordinário.

E a noção de beleza do Ward? Todos já reclamamos do fato de nossas linguagens não tiverem sido desenvolvidas para os nossos problemas. Mas a declaração de Ward coloca novamente o peso sobre nós. Ele diz que um código belo *faz parecer que a linguagem foi feita para o problema!* Portanto, é *nossa* responsabilidade fazer a linguagem parecer simples! Há brigas por causa das linguagens em todo lugar, cuidado! Não é a linguagem que faz os programas parecerem simples, é o programador!

Escolas de Pensamento

E eu (Tio Bob)? O que é um código limpo para mim? É isso o que este livro lhe dirá em detalhes, o que eu e meus compatriotas consideramos um código limpo. Diremo-lhe o que consideramos como nomes limpos de variáveis, funções limpas, classes limpas etc. Apresentaremos nossos conceitos como verdades absolutas, e não nos desculparemos por nossa austeridade. Para nós, a essa altura de nossa carreira, tais conceitos *são absolutos*. São *nossa escola de pensamento* acerca do que seja um código limpo.

Nem todos os mestres de artes marciais concordam com qual seria a melhor arte marcial de todas ou a melhor técnica dentro de uma arte marcial específica. Geralmente, eles criam suas próprias escolas de pensamento e recrutam alunos para serem ensinados. Dessa forma, temos o *Jiu Jitsu dos Gracie*, criado e ensinado pela família Gracie, no Brasil; o *Jiu Jitsu de Hakkoryu*, criado e ensinado por Okuyama Ryuho, em Tóquio; e o *Jeet Kune Do*, criado e ensinado por Bruce Lee, nos EUA.

Os estudantes se dedicam à doutrina ensinada por aquela determinada arte, e aprendem o que seus mestres ensinam, geralmente ignorando os ensinamentos de outros mestres. Depois, conforme os estudantes progridem, costuma-se mudar o mestre de modo que possam expandir seus conhecimentos e práticas. Alguns, no final, continuam a fim de refinar suas habilidades, descobrem novas técnicas e criam suas próprias escolas.

Nenhuma dessas escolas está 100% *certa*. Mesmo assim dentro de uma determinada escola *agimos* como os ensinamentos e técnicas *fossem* os certos. Apesar de tudo, há uma forma correta de praticar o *Jiu Jitsu de Hakkoryu*, ou *Jeet Kune Do*. Mas essa retidão dentro de uma escola não invalida as técnicas de outra.

Considere este livro como uma descrição da *Escola de Código Limpo da Object Mentor*. As técnicas e ensinamentos são a maneira pela qual praticamos nossa *arte*. Estamos dispostos a alegar que se você seguir esses ensinamentos, desfrutará dos benefícios que também aproveitamos e aprenderá a escrever códigos limpos e profissionais. Mas não pense você que nós estamos 100% “certos”. Provavelmente há outras escolas e mestres que têm tanto para oferecer quanto nós. O correto seria que você aprendesse com elas também.

De fato, muitas das recomendações neste livro são contraditórias. Provavelmente você não concordará com todas e poderá até mesmo discordar intensivamente com algumas. Tudo bem. Não podemos querer ter a palavra final. Por outro lado, pensamos bastante e por muito tempo sobre as recomendações neste livro. As aprendemos ao longo de décadas de experiência e repetidos testes e erros. Portanto, concorde você ou não, seria uma pena se você não conseguisse enxergar nosso ponto de vista.



Somos Autores

O campo `@author` de um Javadoc nos diz quem somos. Nós somos autores, e todo autor tem leitores, com os quais uma boa comunicação é de responsabilidade dos autores. Na próxima vez em que você escrever uma linha de código, lembre-se de que você é um autor, escrevendo para leitores que julgarão seus esforços.

Você talvez se pergunte: o quanto realmente se lê de um código? A maioria do trabalho não é escrevê-lo?

Você já reproduziu uma sessão de edição? Nas décadas de 1980 e 1990, tínhamos editores, como o Emacs, que mantinham um registro de cada tecla pressionada. Você podia trabalhar por horas e, então, reproduzir toda a sua sessão de edição como um filme passando em alta velocidade. Quando fiz isso, os resultados foram fascinantes.

A grande maioria da reprodução era rolamento de tela e navegação para outros módulos!

Bob entra no modulo.

Ele descia até a função que precisava ser alterada.

Ele para e pondera suas opções.

Oh, ele sobe para o inicio do módulo a fim de verificar a inicialização da variável.

Agora ele desce novamente e começa a digitar.

Opa, ele está apagando o que digitou!

Ele digita novamente.

Ele apaga novamente.

Ele digita a metade de algo e apaga!

Ele desce até outra função que chama a que ele está modificando para ver como ela é chamada.

Ele sobe novamente e digita o mesmo código que acabara de digitar.

Ele para.

Ele apaga novamente!

Ele abre uma outra janela e analisa uma subclasse. A função é anulada?

...

Bem, você entendeu! Na verdade, a taxa de tempo gasto na leitura v. na escrita é de 10x1.

Constantemente lemos um código antigo quando estamos criando um novo.

Devido à tamanha diferença, desejamos que a leitura do código seja fácil, mesmo se sua criação for árdua. É claro que não há como escrever um código sem lê-lo, portanto torná-lo de fácil leitura realmente facilita a escrita.

Não há como escapar desta lógica. Você não pode escrever um código se não quiser ler as outras partes dele. O código que você tenta escrever hoje será de difícil ou fácil leitura dependendo da facilidade de leitura da outra parte do código. Se quiser ser rápido, se quiser acabar logo, se quiser que seu código seja de fácil escrita, torne-o de fácil leitura.

A Regra de Escoteiro

Não basta escrever um código bom. *Ele precisa ser mantido sempre limpo.* Todos já vimos códigos estragarem e degradarem com o tempo. Portanto, precisamos assumir um papel ativo na prevenção da degradação.

A *Boy Scouts of America* (maior organização de jovens escoteiros dos EUA) tem uma regra simples que podemos aplicar à nossa profissão.

Deixe a área do acampamento mais limpa do que como você a encontrou.⁵

Se todos deixássemos nosso código mais limpo do que quando o começamos, ele simplesmente não degradaria. A limpeza não precisa ser algo grande. Troque o nome de uma variável por um melhor, divida uma função que esteja um pouco grande demais, elimine um pouco de repetição de código, reduza uma instrução if aninhada.

Consegue se imaginar trabalhando num projeto no qual o código *simplesmente melhorou* com o tempo? Você acredita que qualquer alternativa seja profissional? Na verdade, o aperfeiçoamento contínuo não é inerente ao profissionalismo?

Prequela e Princípios

Em muitas maneiras este livro é uma “prequela” de outro que escrevi em 2002, chamado *Agile Software Development: Principles, Patterns, and Practices* (PPP). Ele fala sobre os princípios do projeto orientado a objeto e muitas das práticas utilizadas por desenvolvedores profissionais. Se você ainda não leu o PPP, talvez ache que é a continuação deste livro. Se já o leu, então perceberá que ele é bastante parecido a esse em relação aos códigos.

Neste livro há referências esporádicas a diversos princípios de projeto, dentre os quais estão: Princípio da Responsabilidade Única (SRP, sigla em inglês), Princípio de Aberto-Fechado (OCP, sigla em inglês), Princípio da Inversão da Independência (DIP, sigla em inglês), dentre outros. Esses princípios são descritos detalhadamente no PPP.

Conclusão

Livros sobre arte não prometem lhe tornar um artista. Tudo o que podem fazer é lhe oferecer algumas das ferramentas, técnicas e linhas de pensamento que outros artistas usaram. Portanto, este livro não pode prometer lhe tornar um bom programador. Ou lhe dar a “sensibilidade ao código”. Tudo o que ele pode fazer é lhe mostrar a linha de pensamento de bons programadores e os truques, técnicas e ferramentas que eles usam.

Assim como um livro sobre arte, este está cheio de detalhes. Há muitos códigos. Você verá códigos bons e ruins; código ruim sendo transformado em bom; listas de heurísticas, orientações e técnicas; e também exemplo após exemplo. Depois disso, é por sua conta.

Lembre-se daquela piada sobre o violinista que se perdeu no caminho para a apresentação em

um concerto? Ele aborda um senhor na esquina e lhe pergunta como chegar ao Carnegie Hall. O senhor observa o violinista com seu violino debaixo dos braços e diz “Pratique, filho. Pratique!”.

Bibliografia

[Beck07]: *Implementation Patterns*, Kent Beck, Addison-Wesley, 2007.

[Knuth92]: Literate Programming, Donald E. Knuth, Center for the Study of Language and Information, Leland Stanford Junior University, 1992.

Introdução ao nomeação de software

Este é o segundo artigo da série "Nomeando seu software". No artigo anterior, eu falei sobre nomes de classes e pacotes, assim como os arquivos-fonte e os diretórios que os possuem. Nomeamos também nossos arquivos jar e war e ear. E nomeamos, nomeamos e nomeamos. Como fazemos muito isso, é melhor que o façamos bem. A seguir estão algumas regras simples para a criação de bons nomes.

2

Nomes Significativos

por Tim Ottinger



Introdução

Há nomes por todos os lados em um software. Nomeamos nossas variáveis, funções, parâmetros, classes e pacotes, assim como os arquivos-fonte e os diretórios que os possuem. Nomeamos também nossos arquivos jar e war e ear. E nomeamos, nomeamos e nomeamos. Como fazemos muito isso, é melhor que o façamos bem. A seguir estão algumas regras simples para a criação de bons nomes.

Use Nomes que Revelem seu Propósito

Dizer que os nomes devem demonstrar seu propósito é fácil. Mas queremos que você saiba que estamos falando *sério*. Escolher bons nomes leva tempo, mas economiza mais. Portanto, cuide de seus nomes e troque-os quando encontrar melhores. Todos que lerem seu código (incluindo você mesmo) ficarão agradecidos.

O nome de uma variável, função ou classe deve responder a todas as grandes questões. Ele deve lhe dizer porque existe, o que faz e como é usado. Se um nome requer um comentário, então ele não revela seu propósito.

```
int d; // tempo decorrido em dias
```

O nome `d` não revela nada. Ele não indica a ideia de tempo decorrido, nem de dias. Devemos escolher um nome que especifique seu uso para mensuração e a unidade usada.

```
int elapsedTimeInDays;
int daysSinceCreation;
int daysSinceModification;
int fileAgeInDays;
```

Escolher nomes que revelem seu propósito pode facilitar bastante o entendimento e a alteração do código. Qual o propósito deste código?

```
public List<int[]> getThem() {
    List<int[]> list1 = new ArrayList<int[]>();
    for (int[] x : theList)
        if (x[0] == 4)
            list1.add(x);
    return list1;
}
```

Por que é difícil dizer o que o código faz? Não há expressões complexas.

O espaçamento e a endentação são cabíveis. Só há três variáveis e duas constantes. Nem mesmo há classes refinadas ou métodos polimórficos, apenas uma lista de vetores (pelo menos é o que parece). O problema não é a simplicidade do código, mas seu aspecto implícito, isto é, o contexto não está explícito no código. Devido a isso, é necessário que saibamos as respostas para questões como:

1. Que tipos de coisas estão em `theList`?
2. Qual a importância de um item na posição zero na `theList`?
3. Qual a importância do valor 4?
4. Como eu usaria a lista retornada?

As respostas para tais questões não estão presentes no exemplo do código, mas poderiam. Digamos que estejamos trabalhando num jogo do tipo “campo minado”. Percebemos que o tabuleiro é uma lista de células chamada `theList`. Vamos renomeá-la para `gameBoard`.

Cada quadrado no tabuleiro é representada por um vetor simples. Mais tarde, descobrimos que a posição zero é armazena um valor de status e que o valor 4 significa “marcado com uma bandeirinha”. Ao dar esses nomes explicativos, podemos melhorar consideravelmente o código:

```
public List<int[]> getFlaggedCells() {  
    List<int[]> flaggedCells = new ArrayList<int[]>();  
    for (int[] cell : gameBoard)  
        if (cell[STATUS_VALUE] == FLAGGED)  
            flaggedCells.add(cell);  
    return flaggedCells;  
}
```

Note que a simplicidade do código não mudou. Ele ainda possui o mesmo número de operadores e constantes, com o mesmo número de itens aninhados. Entretanto, o código ficou muito mais explícito.

Podemos continuar e criar uma classe simples para as células, em vez de usar um vetor `int`s.

Ela pode ter uma função com um nome que revele seu propósito (chamada `isFlagged`, ou seja “está marcada com uma bandeirinha”) para ocultar os números mágicos.

O resultado é uma nova versão da função:

```
public List<Cell> getFlaggedCells() {  
    List<Cell> flaggedCells = new ArrayList<Cell>();  
    for (Cell cell : gameBoard)  
        if (cell.isFlagged())  
            flaggedCells.add(cell);  
    return flaggedCells;  
}
```

Com essas simples alterações de nomes, não fica difícil entender o que está acontecendo. Esse é o poder de escolher bons nomes.

Evite Informações Erradas

Os programadores devem evitar passar dicas falsas que confundam o sentido do código. Devemos evitar palavras cujos significados podem se desviar daquele que desejamos. Por exemplo, `hp`, `aix` e `sco` seriam nomes ruins de variáveis, pois são nomes de plataformas Unix ou variantes. Mesmo se estiver programando uma hipotenusa e `hp` parecer uma boa abreviação, o nome pode ser mal interpretado.

Não se refira a um grupo de contas como `accountList`, a menos que realmente seja uma `List`. A palavra *list* (lista) significa algo específico para programadores. Se o que armazena as contas não for uma `List` de verdade, poderá confundir os outros.¹ Portanto, `accountGroup` ou `bunchOfAccounts` ou apenas `accounts` seria melhor.

Cuidado ao usar nomes muito parecidos. Fica difícil perceber a pequena diferença entre `XYZControllerForEfficientHandlingOfStrings` em um módulo e `XYZControllerForEfficientStorageOfStrings` em outro. Ambas as palavras são muito semelhantes.

¹ Como vimos de vez, mesmo se for uma lista, tecnicamente é melhor não colocar o tipo no nome.

Usar conceitos similares para montar palavras é *informação*. Usar formatos inconsistentes para palavras leva a uma *má interpretação*. Com os ambientes Java de hoje dispomos do recurso de autocompletar palavras. Digitamos alguns caracteres de um nome e pressionamos uma combinação de teclas (se houver uma) e, então, aparece uma lista de possíveis palavras que se iniciam com tais letras. Isso é muito prático se houver nomes muito parecidos organizados alfabeticamente num mesmo local e se as diferenças forem óbvias, pois é mais provável que o desenvolvedor escolha um objeto pelo nome, sem consultar seus comentários ou mesmo a lista de métodos fornecidos por aquela classe.

Um exemplo real de nomes que podem gerar confusão é o uso da letra “l” minúscula ou da vogal “O” maiúscula como nome de variáveis. O problema é que eles se parecem com o um e o zero, respectivamente.

```
int a = 1;
if ( O == 1 )
    a = 01;
else
    l = 01;
```

O leitor pode achar que inventamos esse exemplo, mas já vimos códigos nos quais isso acontecia bastante. Uma vez, o autor do código sugeriu o uso de fontes distintas de modo a realçar mais as diferenças, uma solução que teria de ser repassada de forma oral ou escrita a todos os futuros desenvolvedores. Com uma simples troca de nome, o problema é resolvido com objetividade e sem precisar criar novas tarefas.

Faça Distinções Significativas

Os programadores criam problemas para si próprios quando criam um código voltado unicamente para um compilador ou interpretador. Por exemplo, como não é possível usar o mesmo nome para referir-se a duas coisas diferentes num mesmo escopo, você pode decidir alterar o nome de maneira arbitrária. Às vezes, isso é feito escrevendo um dos nomes errado, produzindo a uma situação na qual a correção de erros de ortografia impossibilita a compilação.² Não basta adicionar números ou palavras muito comuns, mesmo que o compilador fique satisfeito. Se os nomes precisam ser diferentes, então também devem ter significados distintos.

Usar números sequenciais em nomes (a1, a2,...,aN) é o oposto da seleção de nomes expressivos. Eles não geram confusão, simplesmente não oferecem informação alguma ou dica sobre a intenção de seu criador. Considere o seguinte:

```
public static void copyChars(char a1[], char a2[]) {
    for (int i = 0; i < a1.length; i++) {
        a2[i] = a1[i];
    }
}
```



Fica muito mais fácil ler essa função quando usam-se source e destination como nomes de parâmetros. Palavras muito comuns são outra forma de distinção que nada expressam. Imagine que você tenha uma classe Product. Se houver outra chamada ProductInfo ou ProductData, você usou nomes distintos que não revelam nada de diferente. Info e Data são palavras muito comuns e vagas, como “um”, “uma” e “a”.

Observe que não há problema em usar prefixos como “um” e “a”, contanto que façam uma distinção significativa. Por exemplo, você pode usar “um” para variáveis locais e “a” para todos os parâmetros de funções.³ O problema surge quando você decide chamar uma variável de aZork só porque já existe outra chamada zork.

Palavras muito comuns são redundantes. O nome de uma variável jamais deve conter a palavra “variável”. O nome de uma tabela jamais deve conter a palavra tabela. Então como NameString é melhor do que Name? Um Name pode ser um número do tipo ponto flutuante? Caso possa, estaria violando uma regra sobre passar informações incorretas. Imagine que você encontre uma classe Customer e outra CustomerObject. O que a diferença nos nomes lhe diz? Qual seria a melhor para possuir o histórico de pagamento de um cliente?

Conhecemos um aplicativo que é assim. A fim de proteger seus desenvolvedores, trocamos os nomes, mas abaixo está exatamente o tipo de erro:

```
getActiveAccount();
getActiveAccounts();
getActiveAccountInfo();
```

Como os programadores desse projeto poderiam saber qual das três funções chamar?

Na ausência de convenções específicas, não há como distinguir moneyAmount de money, customerInfo de customer, accountData de account e theMessage de message. Faça a distinção dos nomes de uma forma que o leitor compreenda as diferenças.

Use Nomes Pronunciáveis

Os ser humano é bom com as palavras. Uma parte considerável de seu cérebro é responsável pelo conceito das palavras. E, por definição, as palavras são pronunciáveis. Seria uma lástima não tirar proveito dessa importante parte de nosso cérebro que evoluiu para lidar com a língua falada. Sendo assim, crie nomes pronunciáveis.

Se não puder pronunciá-lo, não terá como discutir sobre tal nome sem parecer um idiota. “Bem, aqui no bê cê erre três cê ene tê, temos um pê esse zê quê int, viram?”. Isso importa porque a programação é uma atividade social.

Conheço uma empresa que possui uma variável genymdhms (*generation date, year, month, day, hour, minute e second*) e seus funcionários saem falando “gê dâbliu eme dê agá eme esse”. Tenho um hábito irritante de pronunciar tudo como está escrito, portanto comecei a falar “gen-yah-muddahims”.

Depois desenvolvedores e analistas estavam falando assim também, e ainda soava estúpido. Mas estávamos fazendo uma brincadeira e, por isso, foi divertido. Engraçado ou não, estamos tolerando nomeações de baixa qualidade. Novos desenvolvedores tiveram de pedir que lhes explicassem as variáveis, e, então, em vez de usar termos existentes na língua, inventavam palavras bobas ao pronunciá-las. Compare

```

class DtaRcrd102 {
    private Date genymdhms;
    private Date modymdhms;
    private final String pszqint = "102";
    /* ... */
}

com

class Customer {
    private Date generationTimestamp;
    private Date modificationTimestamp;;
    private final String recordId = "102";
    /* ... */
}

```

Agora é possível uma conversa inteligente: “Ei, Mikey, dê uma olhada neste registro! A *generation timestamp* (“criação de marcação de horário”) está marcada para amanhã! Como pode?”.

Use Nomes Passíveis de Busca

Nomes de uma só letra ou números possuem um problema em particular por não ser fácil localizá-los ao longo de um texto. Pode-se usar facilmente o grep para MAX_CLASSES_PER_STUDENT, mas buscar o número 7 poderia ser mais complicado. Nomes, definições de constantes e várias outras expressões que possuam tal número, usado para outros propósitos podem ser resultados da busca. Pior ainda quando uma constante é um número grande e alguém talvez tenha trocado os dígitos, criando assim um bug e ao mesmo tempo não sendo captada pela busca efetuada.

Da mesma forma, o nome “e” é uma escolha ruim para qualquer variável a qual um programador talvez precise fazer uma busca. É uma letra muito comum e provavelmente aparecerá em todo texto em qualquer programa. Devido a isso, nomes longos se sobressaem aos curtos, e qualquer nome passível de busca se sobressai a uma constante no código.

Particularmente, prefiro que nomes com uma única letra SÓ sejam usados como variáveis locais dentro de métodos pequenos. *O tamanho de um nome deve ser proporcional ao tamanho do escopo.*

[N5]. Se uma variável ou constante pode ser vista ou usada em vários lugares dentro do código, é imperativo atribuí-la um nome fácil para busca. Novamente, compare

```

for (int j=0; j<34; j++) {
    s += (t[j]*4)/5;
}

com

int realDaysPerIdealDay = 4;
const int WORK_DAYS_PER_WEEK = 5;
int sum = 0;
for (int j=0; j < NUMBER_OF_TASKS; j++) {
    int realTaskDays = taskEstimate[j] * realDaysPerIdealDay;
    int realTaskWeeks = (realdays / WORK_DAYS_PER_WEEK);
    sum += realTaskWeeks;
}

```

Note que `sum` não é um nome prático, mas pelo menos é fácil de procurar. O nome usado no código serve para uma função maior, mas pense como seria muito mais fácil encontrar `WORK_DAYS_PER_WEEK` do que buscar em todos os lugares nos quais o `5` aparece e, então, filtrar os resultados para exibir apenas as instâncias que você deseja.

Evite Codificações

Já temos de lidar com bastante codificação e não precisamos acrescentar mais. Codificar informações do escopo ou tipos em nomes simplesmente adiciona uma tarefa extra de decifração. Dificilmente parece lógico contratar um novo funcionário para aprender outra “linguagem” codificadora além da atual usada no código no qual se está trabalhando. É uma sobrecarga mental desnecessária ao tentar resolver um problema. Nomes codificados raramente são pronunciáveis, além de ser fácil escrevê-los incorretamente.

A Notação Húngara

Antigamente, quando trabalhávamos com linguagens com limite de tamanho para os nomes, violávamos essa regra quando necessário, com certo arrependimento. O Fortran forçava codificações ao tornar a primeira letra uma indicação para o tipo. Versões anteriores do BASIC só permitiam uma letra mais um dígito. A Notação Húngara (NH) inovou essas limitações. Na época da API em C do Windows, a NH era considerada muito importante, quando tudo era um inteiro ou um ponteiro para um inteiro long de 32 bits ou um ponteiro do tipo void, ou uma das diversas implementações de “strings” (com finalidades e atributos diferentes). O compilador não verificava os tipos naquele tempo, então os programadores precisavam de ajuda para lembrar dos tipos.

Em linguagens modernas, temos tipos muito melhores, e os compiladores os reconhecem e os tornam obrigatórios. Ademais, há uma certa tendência para a criação de classes e funções menores, de modo que as pessoas possam ver onde cada variável que estão usando foi declarada.

Os programadores Java não precisam definir o tipo. Os objetos já são o próprio tipo, e a edição de ambientes se tornou tão avançada que detectam quando se usa inadequadamente um tipo antes mesmo da compilação! Portanto, hoje em dia, a NH e outras formas de convenção de tipos são basicamente obstáculos. Eles dificultam a alteração do nome ou do tipo de uma variável, função ou classe; dificultam a leitura do código; e criam a possibilidade de que o sistema de codificação induza o leitor ao erro.

```
PhoneNumber phoneString;  
// o nome não muda na alteração do tipo!
```

Prefixos de Variáveis Membro

Você não precisa mais colocar o prefixo `m_` em variáveis membro. Mas para isso, suas classes e funções devem ser pequenas. E você deve usar um ambiente de edição que realce ou colore as variáveis membro de modo a distingui-las.

```
public class Part {  
    private String m_dsc; // Descrição textual  
    void setName(String name) {
```

```
m_dsc = name;  
}  
}  
  
public class Part {  
    String description;  
    void setDescription(String description) {  
        this.description = description;  
    }  
}
```

Além disso, as pessoas rapidamente aprendem a ignorar o prefixo (ou sufixo) para visualizar a parte significativa do nome. Quanto mais lemos o código, menos prefixos enxergamos. No final, estes se tornam partes invisíveis e um indicativo de código velho.

Interfaces e Implementações

Às vezes, há casos especiais para codificações. Por exemplo, digamos que você esteja construindo uma ABSTRACT FACTORY para criar formas. Essa factory será uma interface, e implementada por uma classe concreta. Como devemos nomeá-la? IShapeFactory e ShapeFactory? Prefiro não enfeitar as interfaces. O “I” no início, tão comum no hoje em dia, é na melhor das hipóteses uma distração, e na pior são informações excessivas. Não quero que meus usuários saibam que estou lhes dando uma interface, e, sim, apenas uma ShapeFactory. Portanto, se eu devo codificar seja a interface ou a implementação, escolho esta. Para codificar a interface, é preferível chamá-la de ShapeFactoryImp, ou mesmo de CShapeFactory.

Evite o Mapeamento Mental

Os leitores não devem ter de traduzir mentalmente os nomes que você escolheu por outros que eles conhecem. Essa questão costuma levar a decisão de não usar os termos do domínio do problema e nem os da solução. Este é um problema com nomes de variáveis de uma só letra. Certamente um contador de iterações pode ser chamado de “i”, “j” ou “k” (mas nunca 1) – isso já se tornou uma tradição – se seu escopo for muito pequeno e não houver outros nomes que possam entrar em conflito com ele.

Entretanto, na maioria dos contextos, um nome de uma só letra é uma escolha ruim; é apenas um armazenador que o leitor deverá mentalmente mapear de acordo com o conceito em uso. Não há razão pior do que usar o nome “c” só porque “a” e “b” já estão sendo usados.

De maneira geral, os programadores são pessoas muito espertas. E esse tipo de pessoas gosta de se exibir mostrando suas habilidades mentais. Apesar de tudo, se você puder confiantemente se lembrar de que o “r” minúsculo é uma versão da url sem o host e o contexto, então obviamente você é muito esperto.

Uma diferença entre um programador esperto e um programador profissional é que este entende que *clareza é fundamental*. Os profissionais usam seus poderes para o bem, e escrevem códigos que outros possam entender.

Nomes de Classes

Classes e objetos devem ter nomes com substantivo(s), como Cliente, PaginaWiki, Conta e AnaliseEndereco. Evite palavras como Gerente, Processador, Dados ou Info no nome de uma classe, que também não deve ser um verbo.

Nomes de Métodos

Os nomes de métodos devem ter verbos, como postarPagamento, excluirPagina ou salvar. Devem-se nomear métodos de acesso, alteração e autenticação segundo seus valores e adicionar os prefixos get, set ou is de acordo com o padrão javabean.⁴

```
string name = employee.getName();
customer.setName("mike");
if (paycheck.isPosted())...
```

Quando os construtores estiverem sobrecarregados, use métodos factory estáticos com nomes que descrevam os parâmetros. Por exemplo,

```
Complex fulcrumPoint = Complex.FromRealNumber(23.0);
```

é melhor do que

```
Complex fulcrumPoint = new Complex(23.0);
```

Para forçar seu uso, torne os construtores correspondentes como privados.

Não dê uma de Espertinho

Se os nomes forem muito “espertinhos”, apenas as pessoas que compartilhem do mesmo senso de humor que seu dono irão lembrá-los, e só enquanto se lembrarem da brincadeira. Eles saberão o que deve fazer a função HolyHandGrenade? Claro, é engraçado, mas talvez nesse caso DeleteItems fique melhor. Opte por clareza no lugar de divertimento.

Essas gracinhias em códigos costumam aparecer na forma de coloquialismos e gírias. Por exemplo, não use firmar() para significar terminar(). Não use piadas de baixo calão, como cairFora para significar abortar(). Diga o que você quer expressar. Expresse o que você quer dizer.



Selecione uma Palavra por Conceito

Escolha uma palavra por cada conceito abstrato e fique com ela. Por exemplo, é confuso ter pegar, recuperar e obter como métodos equivalentes de classes diferentes. Como lembrar a qual método pertence cada classe? Infelizmente, você geralmente precisa se lembrar qual empresa, grupo ou pessoa criou a biblioteca ou a classe de modo a recordar qual termo foi usado. Caso contrário, você perde muito tempo vasculhando pelos cabeçalhos e exemplos de códigos antigos.

Os ambientes modernos de edição, como o Eclipse e o IntelliJ, oferecem dicas relacionadas ao contexto, como a lista de métodos que você pode chamar em um determinado objeto. Mas note que a lista geralmente não lhe oferece os comentários que você escreveu em torno dos nomes de suas funções. Você tem sorte se receber o parâmetro *nomes* (*names*) das declarações das funções. Os nomes das funções têm de ficar sozinhos, e devem ser consistentes de modo que você possa selecionar o método correto sem qualquer busca extra.

Da mesma forma, é confuso ter um controlador, um gerenciador e um driver no mesmo código-fonte. Qual a principal diferença entre um GerenciadorDeDispositivo e um controlador-de-protocolo? Por que ambos não são controladores ou gerenciadores? Ambos são realmente drivers? O nome faz com que você espere dois objetos com tipos bem distintos, assim como ter classes diferentes.

Um léxico consistente é uma grande vantagem aos programadores que precisem usar seu código.

Não Faça Trocadilhos

Evite usar a mesma palavra para dois propósitos. Usar o mesmo termo para duas ideias diferentes é basicamente um trocadilho.

Se você seguir a regra “uma palavra por conceito”, você pode acabar ficando com muitas classes que possuam, por exemplo, um método add. Contanto que as listas de parâmetros e os valores retornados dos diversos métodos add sejam semanticamente equivalentes, tudo bem.

Entretanto, uma pessoa pode decidir usar a palavra add por fins de “consistência” quando ela na verdade não aplica o mesmo sentido a todas. Digamos que tenhamos muitas classes nas quais add criará um novo valor por meio da adição e concatenação de dois valores existentes. Agora, digamos que estejamos criando uma nova classe que possua um método que coloque seu único parâmetro em uma coleção. Deveríamos chamar este método de add? Por termos tantos outros métodos add, isso pode parecer consistente. Mas, neste caso, a semântica é diferente. Portanto, deveríamos usar um nome como inserir ou adicionar. Chamar este novo método de add seria um trocadilho.

Nosso objetivo, como autores, é tornar a leitura de nosso código o mais fácil possível. Desejamos que nosso código seja de rápida leitura, e não um estudo demorado. Queremos usar a linguagem de um livro popular no qual é responsabilidade do autor ser claro, e não uma linguagem acadêmica na qual a tarefa do estudioso é entender minuciosamente o que está escrito.

Use nomes a partir do Domínio da Solução

Lembre-se de que serão programadores que lerão seu código. Portanto, pode usar termos de Informática, nomes de algoritmos, nomes de padrões, termos matemáticos etc. Não é prudente

pensar num nome a partir do domínio do problema, pois não queremos que nossos companheiros de trabalho tenham de consultar o cliente toda hora para saber o significado de um nome o qual eles já conhecem o conceito, só que por outro nome.

O nome `AccountVisitor` (“conta do visitante”) significa o bastante para um programador familiarizado com o padrão `VISITOR`. Qual programador não saberia o que é uma `JobQueue` (“fila de tarefas”)? Há muitas coisas técnicas que os programadores devem fazer. Selecionar nomes técnicos para tais coisas é, geralmente, o método mais adequado.

Use nomes de Domínios do Problema

Quando não houver uma solução “à la programador”, use o nome do domínio do problema. Pelo menos o programador que fizer a manutenção do seu código poderá perguntar a um especialista em tal domínio o que o nome significa.

Distinguir os conceitos do domínio do problema dos do domínio da solução é parte da tarefa de um bom programador e designer. O código que tem mais a ver com os conceitos do domínio do problema tem nomes derivados de tal domínio.

Adicione um Contexto Significativo

Há poucos nomes que são significativos por si só—a maioria não é. Por conta disso, você precisa usar nomes que façam parte do contexto para o leitor. Para isso você os coloca em classes, funções e namespaces bem nomeados. Se nada disso funcionar, então talvez como último recurso seja necessário adicionar prefixos ao nome.

Imagine que você tenha variáveis chamadas `firstName`, `lastName`, `street`, `houseNumber`, `city`, `state` e `zipcode`. Vistas juntas, fica bem claro que elas formam um endereço. Mas e se você só visse a variável `state` sozinha num método? Automaticamente você assumiria ser parte de um endereço?

Podem-se usar prefixos para adicionar um contexto: `addrFirstName`, `addrLastName`, `addrState` etc. Pelo menos os leitores entenderão que essas variáveis são parte de uma estrutura maior. É claro que uma melhor solução seria criar uma classe chamada `Address`. Então, até o compilador sabe que as variáveis pertencem a um escopo maior.

Veja o método na Listagem 2.1. As variáveis precisam de um contexto mais significativo? O nome da função oferece apenas parte do contexto; o algoritmo apresenta o resto.

Após ter lido a função, você vê que três variáveis, `number`, `verb` e `pluralModifier`, fazem parte da mensagem de dedução (*guess statistics message*). Infelizmente, o contexto deve ser inferido. Ao olhar pela primeira vez o método, o significado das variáveis não está claro.

Listagem 2-1**Variáveis com contexto obscuro**

```
private void printGuessStatistics(char candidate, int count) {
    String number;
    String verb;
    String pluralModifier;
    if (count == 0) {
        number = "no";
        verb = "Existem";
        pluralModifier = "s";
    } else if (count == 1) {
        number = "1";
        verb = "Existe";
        pluralModifier = "";
    } else {
        number = Integer.toString(count);
        verb = "Existem";
        pluralModifier = "s";
    }
    String guessMessage = String.format(
        "There %s %s %s%s", verb, number, candidate, pluralModifier
    );
    print(guessMessage);
}
```

A função é um pouco extensa demais também e as variáveis são bastante usadas. A fim de dividir a função em partes menores, precisamos criar uma classe `GuessStatisticsMessage` e tornar as três variáveis como campos desta classe. Isso oferecerá um contexto mais claro para as três variáveis. Elas são *definitivamente* parte da `GuessStatisticsMessage`. A melhora do contexto também permite ao algoritmo ficar muito mais claro ao dividi-lo em funções menores (veja a Listagem 2.2).

Listagem 2-2**Variáveis possuem contexto**

```
public class GuessStatisticsMessage {
    private String number;
    private String verb;
    private String pluralModifier;

    public String make(char candidate, int count) {
        createPluralDependentMessageParts(count);
        return String.format(
            "There %s %s %s%s",
            verb, number, candidate, pluralModifier
        );
    }

    private void createPluralDependentMessageParts(int count) {
        if (count == 0) {
            thereAreNoLetters();
        } else if (count == 1) {
            thereIsOneLetter();
        } else {
            thereAreMoreThanOneLetter();
        }
    }
}
```

Listagem 2-2 (continuação)**Variáveis possuem contexto**

```
    thereAreManyLetters(count);
}
}

private void thereAreManyLetters(int count) {
    number = Integer.toString(count);
    verb = "Existem";
    pluralModifier = "s";
}

private void thereIsOneLetter() {
    number = "1";
    verb = "Existe";
    pluralModifier = "";
}
private void thereAreNoLetters() {
    number = "no";
    verb = "Existem";
    pluralModifier = "s";
}
}
```

Não Adicione Contextos Desnecessários

Em um aplicativo fictício chamado “Gas Station Deluxe” (GSD), seria uma péssima ideia adicionar prefixos a toda classe com GSD. Para ser sincero, você estará trabalhando contra suas ferramentas. Você digita G e pressiona a tecla de autocompletar e recebe uma lista quilométrica de cada classe no sistema. Isso é inteligente? Para que dificultar a ajuda da IDE?

Da mesma forma, digamos que você inventou uma classe `MailingAddress` no módulo de contabilidade do GSD e que o chamou de `GSDAccountAddress`. Mais tarde, você precisa armazenar um endereço postal de seu cliente no aplicativo. Você usaria `GSDAccountAddress`? Parece que o nome é adequado? Dez dos 17 caracteres são redundantes ou irrelevantes.

Nomes curtos geralmente são melhores contanto que sejam claros. Não adicione mais contexto a um nome do que o necessário.

Os nomes `accountAddress` e `customerAddress` estão bons para instâncias da classe `Address`, mas seriam ruins para nomes de classes. `Address` está bom para uma classe. Se precisar diferenciar entre endereços MAC, endereços de portas e endereços da Web, uma ideia seria `PostalAddress`, `MAC` e `URI`. Os nomes resultantes são mais precisos, motivo esse da tarefa de se atribuir nomes.

Conclusão

O mais difícil sobre escolher bons nomes é a necessidade de se possuir boas habilidades de descrição e um histórico cultural compartilhado. Essa é uma questão de aprender, e não técnica, gerencial ou empresarial. Como consequência, muitas pessoas nessa área não aprendem essa tarefa muito bem.

Elas também têm receio de renomear as coisas por temer que outros desenvolvedores sejam contra. Não compartilhamos desse medo e achamos que ficamos realmente agradecidos quando os nomes mudam (para melhor). Na maioria das vezes, não memorizamos os nomes de classes e métodos. Mas usamos ferramentas modernas para lidar com detalhes de modo que possamos nos focalizar e ver se o código é lido como parágrafos, frases, ou pelo menos como tabelas e estruturas de dados (uma frase nem sempre é a melhor forma de se exibir dados). Provavelmente você acabará surpreendendo alguém quando renomear algo, assim como qualquer outra melhoria no código. Não deixe que isso atrapalhe seu progresso.

Siga alguma dessas regras e note se você não melhorou a legibilidade de seu código. Se estiver fazendo a manutenção do código de outra pessoa, use ferramentas de refatoração para ajudar a resolver essas questões. Em pouco tempo valerá a pena, e continuará a vale em longo prazo.

3

Funções



Nos primórdios da programação, formávamos nossos sistemas com rotinas e sub-rotinas. Já na era do Fortran e do PL/I, usávamos programas, subprogramas e funções. De tudo isso, apenas função prevaleceu. As funções são a primeira linha de organização em qualquer programa. Escrevê-las bem é o assunto deste capítulo.

Veja o código na Listagem 3.1. É difícil encontrar uma função grande em FitNesse¹, mas procurando um pouco mais encontramos uma. Além de ser longo, seu código é repetido, há diversas strings estranhas e muitos tipos de dados e APIs esquisitos e nada óbvios. Veja o quanto você consegue compreender nos próximos três minutos.

Listagem 3-1**HtmlUtil.java (FitNesse 20070619)**

```
public static String testableHtml(
    PageData pageData,
    boolean includeSuiteSetup
) throws Exception {
    WikiPage wikiPage = pageData.getWikiPage();
    StringBuffer buffer = new StringBuffer();
    if (pageData.hasAttribute("Test")) {
        if (includeSuiteSetup) {
            WikiPage suiteSetup =
                PageCrawlerImpl.getInheritedPage(
                    SuiteResponder.SUITE_SETUP_NAME, wikiPage
                );
            if (suiteSetup != null) {
                WikiPagePath pagePath =
                    suiteSetup.getPageCrawler().getFullPath(suiteSetup);
                String pagePathName = PathParser.render(pagePath);
                buffer.append("!include -setup .")
                    .append(pagePathName)
                    .append("\n");
            }
        }
        WikiPage setup =
            PageCrawlerImpl.getInheritedPage("SetUp", wikiPage);
        if (setup != null) {
            WikiPagePath setupPath =
                wikiPage.getPageCrawler().getFullPath(setup);
            String setupPathName = PathParser.render(setupPath);
            buffer.append("!include -setup .")
                .append(setupPathName)
                .append("\n");
        }
    }
    buffer.append(pageData.getContent());
    if (pageData.hasAttribute("Test")) {
        WikiPage teardown =
            PageCrawlerImpl.getInheritedPage("TearDown", wikiPage);
        if (teardown != null) {
            WikiPagePath tearDownPath =
                wikiPage.getPageCrawler().getFullPath(teardown);
            String tearDownPathName = PathParser.render(tearDownPath);
            buffer.append("\n")
                .append("!include -teardown .")
                .append(tearDownPathName)
                .append("\n");
        }
    }
}
```

Listagem 3-1 (continuação)**HtmlUtil.java (FitNesse 20070619)**

```
if (includeSuiteSetup) {
    WikiPage suiteTeardown =
        PageCrawlerImpl.getInheritedPage(
            SuiteResponder.SUITE_TEARDOWN_NAME,
            wikiPage
        );
    if (suiteTeardown != null) {
        WikiPagePath pagePath =
            suiteTeardown.getPageCrawler().getFullPath(suiteTeardown);
        String pagePathName = PathParser.render(pagePath);
        buffer.append("!include -teardown .")
            .append(pagePathName)
            .append("\n");
    }
}
pageData.setContent(buffer.toString());
return pageData.getHtml();
}
```

Conseguiu entender a função depois desses três minutos estudando-a? Provavelmente não. Há muita coisa acontecendo lá em muitos níveis diferentes de . Há strings estranhas e chamadas a funções esquisitas misturadas com dois `if` aninhados controlados por flags.

Entretanto, com umas poucas extrações simples de métodos, algumas renomeações e um pouco de reestruturação,fui capaz de entender o propósito da função nas nove linhas da Listagem 3.2. Veja se você consegue compreender também em três minutos.

Listagem 3-2**HtmlUtil.java (refatorado)**

```
public static String renderPageWithSetupsAndTeardowns(
    PageData pageData, boolean isSuite
) throws Exception {
    boolean isTestPage = pageData.hasAttribute("Test");
    if (isTestPage) {
        WikiPage testPage = pageData.getWikiPage();
        StringBuffer newPageContent = new StringBuffer();
        includeSetupPages(testPage, newPageContent, isSuite);
        newPageContent.append(pageData.getContent());
        includeTeardownPages(testPage, newPageContent, isSuite);
        pageData.setContent(newPageContent.toString());
    }
    return pageData.getHtml();
}
```

A menos que já estivesse estudando o FitNesse, provavelmente você não entendeu todos os detalhes.

Ainda assim você talvez tenha compreendido que essa função efetua a inclusão de algumas páginas SetUp e TearDown em uma página de teste e, então, exibir tal página em HTML. Se estiver familiarizado com o JUnit², você já deve ter percebido que essa função pertence a algum tipo de framework de teste voltado para a Web. E você está certo. Deduzir tal informação da Listagem 3.2 é muito fácil, mas ela está bastante obscura na Listagem 3.1.

Então, o que torna uma função como a da Listagem 3.2 fácil de ler e entender? Como fazer uma função transmitir seu propósito? Quais atributos dar às nossas funções que permitirão um leitor comum deduzir o tipo do programa ali contido?

Pequenas!

A primeira regra para funções é que elas devem ser pequenas. A segunda é que *precisam ser mais espertas do que isso*. Não tenho como justificar essa afirmação. Não tenho referências de pesquisas que mostrem que funções muito pequenas são melhores. Só posso dizer que por cerca de quatro décadas tenho criado funções de tamanhos variados. Já escrevi diversos monstros de 3.000 linhas; bastantes funções de 100 a 300 linhas; e funções que tinham apenas de 20 a 30 linhas. Essa experiência me ensinou que, ao longo de muitas tentativas e erros, as funções devem ser muito pequenas.

Na década de 1980, costumávamos dizer que uma função não deveria ser maior do que a tela.

É claro que na época usávamos as telas VT100, de 24 linhas por 80 colunas, e nossos editores usavam 4 linhas para fins gerenciamento. Hoje em dia, com fontes reduzidas e um belo e grande monitor, você consegue colocar 150 caracteres em uma linha – não se deve ultrapassar esse limite—e umas 100 linhas ou mais por tela—as funções não devem chegar a isso tudo, elas devem ter no máximo 20 linhas.

O quanto pequena deve ser uma função? Em 1999, fui visitar Kent Beck em sua casa, em Oregon, EUA. Sentamo-nos e programamos um pouco juntos. Em certo ponto, ele me mostrou um simpático programa de nome Java/Swing o qual ele chamava de *Sparkle*. Ele produzia na tela um efeito visual similar a uma varinha mágica da fada madrinha do filme da Cinderela. Ao mover o mouse, faíscas (*sparkles*, em inglês) caíam do ponteiro do mouse com um belo cintilar até o fim da janela, como se houvesse gravidade na tela. Quando Kent me mostrou o código, fiquei surpreso com tamanho pequenezas das funções. Eu estava acostumado a funções que seguiam por quilômetros em programas do Swing. Cada função *neste* programa tinha apenas duas, ou três, ou quatro linhas. Esse deve ser o tamanho das suas funções³.

O quanto pequenas devem ser suas funções? Geralmente menores do que a da Listagem 3.2! Na verdade, a Listagem 3.2 deveria ser enxuta para a Listagem 3.3.

2. Ferramenta de código aberto de teste de unidade para Java. www.junit.org.

3. Quando eu vivia em São Paulo, havia uma loja de informática que vendeu muitos computadores antigos, mas foi em vão.

Listagem 3-3**HtmlUtil.java (refatorado novamente)**

```
public static String renderPageWithSetupsAndTeardowns(
    PageData pageData, boolean isSuite) throws Exception {
    if (isTestPage(pageData))
        includeSetupAndTeardownPages(pageData, isSuite);
    return pageData.getHtml();
}
```

Blocos e Endentação

Aqui quero dizer que blocos dentro de instruções `if`, `else`, `while` e outros devem ter apenas uma linha. Possivelmente uma chamada de função. Além de manter a função pequena, isso adiciona um valor significativo, pois a função chamada de dentro do bloco pode receber um nome descritivo. Isso também implica que as funções não devem ser grandes e ter estruturas aninhadas. Portanto, o nível de endentação de uma função deve ser de, no máximo, um ou dois. Isso, é claro, facilita a leitura e compreensão das funções.

Faça Apenas uma Coisa

Deve ter ficado claro que a Listagem 3.1 faz muito mais de uma coisa. Ela cria buffers, pega páginas, busca por páginas herdadas, exibe caminhos, anexa strings estranhas e gera HTML, dentre outras coisas. A Listagem 3.1 vive ocupada fazendo diversas coisas diferentes. Por outro lado, a Listagem 3.3 faz apenas uma coisa simples. Ela inclui `SetUp` e `TearDown` em páginas de teste.

O conselho a seguir tem aparecido de uma forma ou de outra por 30 anos ou mais.

***AS FUNÇÕES DEVEM FAZER UMA COISA. DEVEM FAZÊ-LA BEM.
DEVEM FAZER APENAS ELA.***



O problema dessa declaração é que é difícil saber o que é “uma coisa”. A Listagem 3.3 faz uma coisa? É fácil dizer que ela faz três:

1. Determina se a página é de teste.
2. Se for, inclui `SetUps` e `TearDowns`.
3. Exibe a página em HTML.

Então, uma ou três coisas? Note que os três passos da função estão em um nível de abaixo do nome da função. Podemos descrever a função com um breve parágrafo TO⁴:

4. A linguagem LOGO usava a palavra “TO” (“PARA”) da mesma forma que Ruby e Python usam “def”. Portanto, toda função começa com a palavra “TO”,

TO `RenderPageWithSetupsAndTeardowns`, verificamos se a página é de teste, se for, incluímos `SetUps` e `TearDowns`. Em ambos os casos, exibimos a página em HTML.

Se uma função faz apenas aqueles passos em um nível abaixo do nome da função, então ela está fazendo uma só coisa. Apesar de tudo, o motivo de criarmos função é para decompor um conceito maior (em outras palavras, o nome da função) em uma série de passos no próximo nível de abstração.

Deve estar claro que a Listagem 3.1 contém passos em muitos níveis diferentes de . Portanto, obviamente ela faz mais de uma coisa. Mesmo a Listagem 3.2 possui dois níveis de abstração , como comprovado pela nossa capacidade de redução. Mas ficaria muito difícil reduzir a Listagem 3.3 de modo significativo. Poderíamos colocar a instrução `if` numa função chamada `includeSetupsAndTeardownsIfTestPage`, mas isso simplesmente reformula o código, sem modificar o nível de .

Portanto, outra forma de saber se uma função faz mais de “uma coisa” é se você pode extrair outra função dela a partir de seu nome que não seja apenas uma reformulação de sua implementação (G34).

Seções Dentro de Funções

Veja a Listagem 4.7 na página 71. Note que a função `generatePrimes` está dividida em seções, como *declarações*, *inicializações* e *seleção*. Esse é um indício óbvio de estar fazendo mais de uma coisa. Não dá para, de forma significativa, dividir em seções as funções que fazem apenas uma coisa.

Um Nível de Abstração por Função

A fim de confirmar se nossas funções fazem só “uma coisa”. Precisamos verificar se todas as instruções dentro da função estão no mesmo nível de abstração. É fácil ver como a Listagem 3.1 viola essa regra. Há outros conceitos lá que estão em um nível de bem alto, como o `getHtml()`; outros que estão em um nível intermediário, como `String pagePathName = PathParser.render(pagePath);` e outros que estão em um nível consideravelmente baixo, como `.append("\n")`.

Vários níveis dentro de uma função sempre geram confusão. Os leitores podem não conseguir dizer se uma expressão determinada é um conceito essencial ou um mero detalhe. Pior, como janelas quebradas, uma vez misturados os detalhes aos conceitos, mais e mais detalhes tendem a se agregar dentro da função.

Ler o Código de Cima para Baixo: *Regra Decrescente*

Queremos que o código seja lido de cima para baixo, como uma narrativa⁵. Desejamos que cada função seja seguida pelas outras no próximo nível de modo que possamos ler o programa descendo um nível de cada vez conforme percorremos a lista de funções. Chamamos isso de *Regra Decrescente*.

Em outras palavras, queremos poder ler o programa como se fosse uma série de parágrafos *TO*, cada um descrevendo o nível atual de e fazendo referência aos parágrafos *TO* consecutivos no próximo nível abaixo.

Para incluir SetUps e TearDowns, incluímos os primeiros, depois o conteúdo da página de teste e, então, adicionamos os segundos. Para incluir SetUps, adicionamos o suite setup, se este for uma coleção, incluímos o setup normal. Para incluir o suite setup, buscamos na hierarquia acima a página “SuiteSetUp” e adicionamos uma instrução de inclusão com o caminho àquela página. Para procurar na hierarquia acima...

Acaba sendo muito difícil para os programadores aprenderem a seguir essa regra e criar funções que fiquem em apenas um nível de . Mas aprender esse truque é também muito importante, pois ele é o segredo para manter as funções curtas e garantir que façam apenas “uma coisa”. Fazer com que a leitura do código possa ser feita de cima para baixo como uma série de parágrafos *TO* é uma técnica eficiente para manter o nível de consistente.

Veja a listagem 3.7 no final deste capítulo. Ela mostra toda a função testableHtml refatorada de acordo com os princípios descrito aqui. Note como cada função indica a seguinte e como cada uma mantém um nível consistente de .

Estrutura Switch

É difícil criar uma estrutura switch pequena⁶, pois mesmo uma com apenas dois cases é maior do que eu gostaria que fosse um bloco ou uma função. Também é difícil construir uma que fala apenas uma coisa. Por padrão, as estruturas switch sempre fazem N coisas. Infelizmente, nem sempre conseguimos evitar o uso do switch, mas podemos nos certificar se cada um está em uma classe de baixo nível e nunca é repetido. Para isso, usamos o polimorfismo.

Veja a Listagem 3.4. Ela mostra apenas uma das operações que podem depender do tipo de funcionário (*employee*, em inglês).

Listagem 3-4

Payroll.java

```
public Money calculatePay(Employee e)
throws InvalidEmployeeType {
    switch (e.type) {
        case COMMISSIONED:
            return calculateCommissionedPay(e);
        case HOURLY:
            return calculateHourlyPay(e);
        case SALARIED:
            return calculateSalariedPay(e);
        default:
            throw new InvalidEmployeeType(e.type);
    }
}
```

Esta função tem vários problemas. Primeiro, ela é grande, e quando se adiciona novos tipos de funcionários ela crescerá mais ainda. Segundo, obviamente ela faz mais de uma coisa. Terceiro, ela viola o Princípio da Responsabilidade Única⁷ (SRP, sigla em inglês) por haver mais de um motivo para alterá-la. Quarto, ela viola o Princípio de Aberto-Fechado⁸ (OCP, sigla em inglês), pois precisa ser modificada sempre que novos tipos forem adicionados. Mas, provavelmente, o pior problema com essa função é a quantidade ilimitada de outras funções que terão a mesma estrutura. Por exemplo, poderíamos ter

```
isPayday(Employee e, Date date)
ou
deliverPay(Employee e, Money pay)
```

ou um outro grupo. Todas teriam a mesma estrutura deletéria.

A solução (veja a Listagem 3.5) é inserir a estrutura switch no fundo de uma ABSTRACT FACTORY⁹ e jamais deixar que alguém a veja. A factory usará o switch para criar instâncias apropriadas derivadas de Employee, e as funções, como calculatePay, isPayday e deliverPay, serão enviadas de forma polifórmica através da interface Employee.

Minha regra geral para estruturas switch é que são aceitáveis se aparecerem apenas uma vez, como para a criação de objetos polifôrmicos, e estiverem escondidas atrás de uma relação de herança de modo que o resto do sistema não possa enxergá-la [G23]. É claro que cada caso é um caso e haverá vezes que não respeitarei uma ou mais partes dessa regra.

Listagem 3-5

Employee e Factory

```
public abstract class Employee {
    public abstract boolean isPayday();
    public abstract Money calculatePay();
    public abstract void deliverPay(Money pay);
}
```

⁷ a. http://en.wikipedia.org/wiki/Single_responsibility_principle

Use Nomes Descritivos

Na Listagem 3.7, eu mudei o nome do exemplo de nossa função `testableHtml` para `SetupTeardownIncluder.render`, que é bem melhor, pois descreve o que a função faz. Também dei a cada método privado nomes igualmente descritivos, como `isTestable` ou `includeSetupAndTeardownPages`. É difícil superestimar o valor de bons nomes. Lembre-se do princípio de Ward: “*Você sabe que está criando um código limpo quando cada rotina que você lê é como você esperava*”. Metade do esforço para satisfazer esse princípio é escolher bons nomes para funções pequenas que fazem apenas uma coisa. Quando menor e mais centralizada for a função, mais fácil será pensar em um nome descritivo.

Não tenha medo de criar nomes extensos, pois eles são melhores do que um pequeno e enigmático. Um nome longo e descritivo é melhor do que um comentário extenso e descritivo. Use uma convenção de nomenclatura que possibilite uma fácil leitura de nomes de funções com várias palavras e, então, use estas para dar à função um nome que explique o que ela faz.

Não se preocupe com o tempo ao escolher um nome. Na verdade, você deve tentar vários nomes e, então, ler o código com cada um deles. IDEs modernas, como Eclipse ou IntelliJ, facilita a troca de nomes. Utilize uma dessas IDEs e experimente diversos nomes até encontrar um que seja bem descritivo.

Selecionar nomes descritivos esclarecerá o modelo do módulo em sua mente e lhe ajudará a melhorá-lo. É comum que ao buscar nomes adequados resulte numa boa reestruturação do código.

Seja consistente nos nomes. Use as mesmas frases, substantivos e verbos nos nomes de funções de seu módulo. Considere, por exemplo, os nomes `includeSetupAndTeardownPages`, `includeSetupPages`, `includeSuiteSetupPage` e `includeSetupPage`. A fraseologia nesses nomes permite uma sequência de fácil dedução. Na verdade, se eu lhe mostrasse apenas a série acima, você se perguntaria: “O que aconteceu com `includeTeardownPages`, `includeSuiteTeardownPage` e `includeTeardownPage`?”, como isso é “... como o que você esperava?”.

Parâmetros de Funções

A quantidade ideal de parâmetros para uma função é zero (nulo). Depois vem um (mônade), seguido de dois (diáde). Sempre que possível devem-se evitar três parâmetros (triade). Para mais de três deve-se ter um motivo muito especial (políade) – mesmo assim não devem ser usados.

Parâmetros são complicados. Eles requerem bastante conceito. É por isso que me livrei de quase todos no exemplo. Considere, por exemplo, o `StringBuffer`. Poderíamos tê-lo passado como parâmetro em vez de instanciá-lo como uma variável, mas então nossos leitores teriam de interpretá-lo sempre que o vissem. Ao ler a



estória contada por pelo módulo, fica mais fácil entender `includeSetupPage()` do que `includeSetupPageInto(newPage-Content)`. O parâmetro não está no nível de que o nome função, forçando-lhe reconhecer de um detalhe (ou seja, o `StringBuffer`) que não seja importante particularmente naquele momento.

Os parâmetros são mais difíceis ainda a partir de um ponto de vista de testes. Imagine a dificuldade de escrever todos os casos de teste para se certificar de que todas as várias combinações de parâmetros funcionem adequadamente. Se não houver parâmetros, essa tarefa é simples. Se houver um, não é tão difícil assim.

Com dois, a situação fica um pouco mais desafiadora. Com mais de dois, pode ser desencorajador testar cada combinação de valores apropriados. Os parâmetros de saída são ainda mais difíceis de entender do que os de entrada. Quando lemos uma função, estamos acostumados à ideia de informações *entrando* na função através de parâmetros e *saindo* através do valor retornado. Geralmente não esperamos dados saindo através de parâmetros. Portanto, parâmetros de saída costumam nos deixar surpresos e fazer com que leiamos novamente.

Um parâmetro de entrada é a melhor coisa depois de zero parâmetro. É fácil entender `SetupTeardown-Includer.render(pageData)`. Está óbvio que renderizemos os dados no objeto `pageData`.

Formas Mônades Comuns

Há duas razões bastante comuns para se passar um único parâmetro a uma função. Você pode estar fazendo uma pergunta sobre aquele parâmetro, como em `boolean fileExists("MyFile")`. Ou você pode trabalhar naquele parâmetro, transformando-o em outra coisa e retornando-o. Por exemplo, `InputStream fileOpen("MyFile")` transforma a `String` do nome de um arquivo em um valor retornado por `InputStream`. São esses dois usos que os leitores esperam ver em uma função.

Você deve escolher nomes que tornem clara a distinção, e sempre use duas formas em um contexto consistente. (Veja a seguir *Separação comando-consulta*). Uma forma menos comum mas ainda bastante útil de um parâmetro para uma função é um *evento*. Nesta forma, há um parâmetro de entrada, mas nenhum de saída. O programa em si serve para interpretar a chamada da função como um evento, e usar o parâmetro para alterar o estado do sistema, por exemplo, `void passwordAttemptFailedNtimes(int attempts)`. Use esse tipo com cautela. Deve ficar claro para o leitor que se trata de um evento. Escolha os nomes e os contextos com atenção.

Tente evitar funções mònades que não sigam essas formas, por exemplo, `void includeSetupPageInto(StringBuffer pageText)`. Usar um parâmetro de saída em vez de um valor de retorno para uma modificação fica confuso. Se uma função vai transformar seu parâmetro de entrada, a alteração deve aparecer como o valor retornado. De fato, `StringBuffer transform(StringBuffer in)` é melhor do que `void transform_(StringBuffer out)`, mesmo que a implementação do primeiro simplesmente retorne o parâmetro de entrada. Pelo menos ele ainda segue o formato de uma modificação.

Parâmetros Lógicos

Esses parâmetros são feios. Passar um booleano para uma função certamente é uma prática horrível, pois ele complica imediatamente a assinatura do método, mostrando explicitamente que a função faz mais de uma coisa. Ela faz uma coisa se o valor for verdadeiro, e outra se for falso!

Na Listagem 3.7, não tínhamos alternativa, pois os chamadores já estavam passando aquela flag (valor booleano) como parâmetro, e eu queria limitar o escopo da refatoração à função e para baixo. Mesmo assim, a chamada do método `render(true)` é muito confusa para um leitor simples. Analisar a chamada e visualizar `render(boolean isSuite)` ajuda um pouco, mas nem tanto. Deveríamos dividir a função em duas: `renderForSuite()` e `renderForSingleTest()`.

Funções Díades

Uma função com dois parâmetros é mais difícil de entender do que uma com um (mônade). Por exemplo, é mais fácil compreender `writeField(name)` do que `writeField(outputStream, name)`¹⁰. Embora o significado de ambas esteja claro, a primeira apresenta seu propósito explicitamente quando a lemos. A segunda requer uma pequena pausa até aprendermos a ignorar o primeiro parâmetro. E isso, é claro, acaba resultando em problemas, pois nunca devemos ignorar qualquer parte do código. O local que ignoramos é justamente aonde se esconderão os bugs.

Há casos, é claro, em que dois parâmetros são necessários como, por exemplo, em `Point p = new Point(0, 0)`. Os pontos de eixos cartesianos naturalmente recebem dois parâmetros. De fato, ficaríamos surpresos se vissemos `new Point(0)`. Entretanto, os dois parâmetros neste caso *são componentes de um único valor!* Enquanto que `outputStream` e `name` não são partes de um mesmo valor.

Mesmo funções díades óbvias, como `assertEquals(expected, actual)`, são problemáticas.

Quantas vezes você já colocou `actual` onde deveria ser `expected`? Os dois parâmetros não possuem uma ordem pré-determinada natural. A ordem `expected, actual` é uma convenção que requer prática para assimilá-la.

Díades não são ruins, e você certamente terá de usá-las. Entretanto, deve-se estar ciente de que haverá um preço a pagar e, portanto, deve-se pensar em tirar proveito dos mecanismos disponíveis a você para convertê-los em mônades. Por exemplo, você poderia tornar o método `writeField` um membro de `OutputStream` de modo que pudesse dizer `outputStream.writeField(name)`; tornar `OutputStream` uma variável membro da classe em uso de modo que não precisasse passá-lo por parâmetro; ou extrair uma nova classe, como `FieldWriter`, que receba o `OutputStream` em seu construtor e possua um método `write`.

Tríades

Funções que recebem três parâmetros são consideravelmente mais difíceis de entender do que as díades. A questão de ordenação, pausa e ignoração apresentam mais do que o dobro de dificuldade. Sugiro que você pense bastante antes de criar uma tríade.

Por exemplo, considere a sobrecarga comum de `assertEquals` que recebe três parâmetros:

`assertEquals(message, expected, actual)`. Quantas vezes você precisou ler o parâmetro `message` e deduzir o que ele carrega? Muitas vezes já me deparei com essa tríade em particular e tive de fazer uma pausa. Na verdade, *toda vez que a vejo*, tenho de ler novamente e, então, a ignoro.

¹⁰ Acabei de refatorar um módulo que usava uma díade. Conseguí tornar o `OutputStream` um campo da classe e converter todas as chamadas ao `writeField` para

formato imbuímos os nomes dos parâmetros no nome da função. Por exemplo, pode ser melhor escrever `assertEquals` do que `assertExpectedEqualsActual(expected, actual)`, o que resolveria o problema de ter de lembrar a ordem dos parâmetros.

Evite Efeitos Colaterais

Efeitos colaterais são mentiras. Sua função promete fazer apenas uma coisa, mas ela também faz outras *coisas escondida*. Às vezes, ela fará alterações inesperadas nas variáveis de sua própria classe. Às vezes, ela adicionará as variáveis aos parâmetros passados à função ou às globais do sistema. Em ambos os casos elas são “verdades” enganosas e prejudiciais, que geralmente resultam em acoplamentos temporários estranhos e dependências.

Considere, por exemplo, a função aparentemente inofensiva na Listagem 3.6. Ela usa um algoritmo padrão para comparar um `userName` (nome de usuário) a um `password` (senha). Ela retorna `true` (verdadeiro) se forem iguais, e `false` (falso) caso contrário. Mas há também um efeito colateral. Consegue identificá-lo?

Listagem 3-6 `UserValidator.java`

```
public class UserValidator {  
    private Cryptographer cryptographer;  
  
    public boolean checkPassword(String userName, String password) {  
        User user = UserGateway.findByName(userName);  
        if (user != User.NULL) {  
            String codedPhrase = user.getPhraseEncodedByPassword();  
            String phrase = cryptographer.decrypt(codedPhrase, password);  
            if ("Valid Password".equals(phrase)) {  
                Session.initialize();  
                return true;  
            }  
        }  
        return false;  
    }  
}
```

O efeito colateral é a chamada ao `Session.initialize()`, é claro. A função `checkPassword`, segundo seu nome, diz que verifica a senha. O nome não indica que ela inicializa a sessão. Portanto, um chamador que acredita no que diz o nome da função corre o risco de apagar os dados da sessão existente quando ele decidir autenticar do usuário.

Esse efeito colateral cria um acoplamento temporário. Isto é, `checkPassword` só poderá ser chamado em determinadas horas (em outras palavras, quando for seguro inicializar a sessão). Se for chamado fora de ordem, sem querer, os dados da sessão poderão ser perdidos. Os acoplamentos temporários são confusos, especialmente quando são um efeito colateral. Se for preciso esse tipo

de acoplamento, é preciso deixar claro no nome da função. Neste caso, poderíamos renomear a função para `checkPasswordAndInitializeSession`, embora isso certamente violaria o “fazer apenas uma única coisa”.

Parâmetros de Saída

Os parâmetros são comumente interpretados como *entradas* de uma função. Se já usa o programa há alguns anos, estou certo de que você já teve de voltar e ler novamente um parâmetro que era, na verdade, de *saída*, e não de entrada. Por exemplo:

```
appendFooter(s);
```

Essa função anexa `s` como rodapé (Footer, em inglês) em algo? Ou anexa um rodapé a `s`? `s` é uma entrada ou uma saída? Não precisa olhar muito a assinatura da função para ver:

```
public void appendFooter(StringBuffer report)
```

Isso esclarece a questão, mas à custa da verificação da declaração da função. Qualquer coisa que lhe force a verificar a assinatura da função é equivalente a uma relida. Isso é uma interrupção do raciocínio e deve ser evitado.

Antes do surgimento da programação orientada a objeto, às vezes era preciso ter parâmetros de saída. Entretanto, grande parte dessa necessidade sumiu nas linguagens OO, pois seu *propósito* é servir como um parâmetro de saída. Em outras palavras, seria melhor invocar `appendFooter` como:

```
report.appendFooter();
```

De modo geral, devem-se evitar parâmetros de saída. Caso sua função precise alterar o estado de algo, faça-a mudar o estado do objeto que a pertence.

Separação comando-consulta

As funções devem fazer ou responder algo, mas não ambos. Sua função ou altera o estado de um objeto ou retorna informações sobre ele. Efetuar as duas tarefas costuma gerar confusão. Considere, por exemplo, a função abaixo:

```
public boolean set(String attribute, String value);
```

Esta função define o valor de um dado atributo e retorna `true` (verdadeiro) se obtiver êxito e `false` (falso) se tal atributo não existir. Isso leva a instruções estranhas como:

```
if (set("username", "unclebob"))...
```

Imagine isso pelo ponto de vista do leitor. O que isso significa? Está perguntando se o atributo “username” anteriormente recebeu o valor “unclebob”? Ou se “username” obteve êxito ao receber o valor “unclebob”? É difícil adivinhar baseando-se na chamada, pois não está claro se a palavra “set” é um verbo ou um adjetivo.

O intuito do autor era que `set` fosse um verbo, mas no contexto da estrutura `if`, parece um adjetivo. Portanto, a instrução lê-se “Se o atributo `username` anteriormente recebeu o valor `unclebob`” e não “atribua o valor `unclebob` ao atributo `username`, e se isso funcionar, então...”. Poderíamos tentar resolver isso renomeando a função `set` para `setAndCheckIfExists`, mas não ajudaria muito para a legibilidade da estrutura `if`.

```
if (attributeExists("username")) {  
    setAttribute("username", "unclebob");  
    ...  
}
```

Prefira exceções a retorno de códigos de erro

Fazer funções retornarem códigos de erros é uma leve violação da separação comando-consulta, pois os comandos são usados como expressões de comparação em estruturas `if`.

```
if (deletePage(page) == E_OK)
```

O problema gerado aqui não é a confusão verbo/adjetivo, mas sim a criação de estruturas aninhadas.

Ao retornar um código de erro, você cria um problema para o chamador, que deverá lidar imediatamente com o erro.

```
if (deletePage(page) == E_OK) {  
    if (registry.deleteReference(page.name) == E_OK) {  
        if (configKeys.deleteKey(page.name.makeKey()) == E_OK){  
            logger.log("página excluída");  
        } else {  
            logger.log("configKey não foi excluída");  
        }  
    } else {  
        logger.log("deleteReference não foi excluído do  
registro");  
    }  
} else {  
    logger.log("a exclusão falhou");  
    return E_ERROR;  
}
```

Por outro lado, se você usar exceções em vez de retornar códigos de erros, então o código de tratamento de erro poderá ficar separado do código e ser simplificado:

```
try {  
    deletePage(page);  
    registry.deleteReference(page.name);  
    configKeys.deleteKey(page.name.makeKey());  
}  
catch (Exception e) {  
    logger.log(e.getMessage());  
}
```

Extraia os blocos try/catch

Esses blocos não têm o direito de serem feios. Eles confundem a estrutura do código e misturam o tratamento de erro com o processamento normal do código. Portanto, é melhor colocar as estruturas try e catch em suas próprias funções.

```
public void delete(Page page) {
    try {
        deletePageAndAllReferences(page);
    }
    catch (Exception e) {
        logError(e);
    }
}

private void deletePageAndAllReferences(Page page) throws Exception {
    deletePage(page);
    registry.deleteReference(page.name);
    configKeys.deleteKey(page.name.makeKey());
}

private void logError(Exception e) {
    logger.log(e.getMessage());
}
```

A função delete acima só faz tratamento de erro. E é fácil entendê-la e seguir adiante. A função deletePageAndAllReferences só trata de processos que excluem toda uma página. Pode-se ignorar o tratamento de erro. Isso oferece uma boa separação que facilita a compreensão e alteração do código.

Tratamento de erro é uma coisa só

As funções devem fazer uma coisa só. Tratamento de erro é uma coisa só. Portanto, uma função que trata de erros não deve fazer mais nada. Isso implica (como no exemplo acima) que a palavra try está dentro de uma função e deve ser a primeira instrução e nada mais deve vir após os blocos catch/finally.

Error.java, o chamariz à dependência

Retornar códigos de erro costuma implicar que há classes ou enum nos quais estão definidos todos os códigos de erro.

```
public enum Error {
    OK,
    INVALID,
    NO_SUCH,
    LOCKED,
    OUT_OF_RESOURCES,
    WAITING_FOR_EVENT;
}
```

Classes como esta são *chamarizes à dependência*, muitas outras classes devem importá-las e usá-las. Portanto, quando o enum da classe Error enum, é preciso recompilar todas as outras classes e redistribuí-las¹¹. Isso coloca uma pressão negativa na classe Error. Os programadores não querem adicionar novos erros porque senão eles teriam de compilar e distribuir tudo novamente. Por isso, eles reutilizam códigos de erros antigos em vez de adicionar novos.

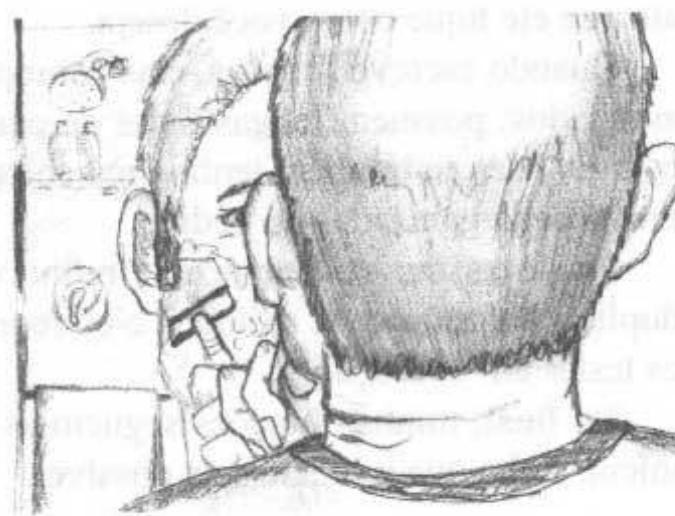
Quando se usam exceções em vez de códigos de erro, as novas exceções são *derivadas* da classe de exceções. Podem-se adicioná-las sem ter de recompilar ou redistribuir¹².

Evite repetição¹³

Leia novamente com atenção a Listagem 3.1 e notará que há um algoritmo que se repete quatro vezes em quatro casos: SetUp, SuiteSetUp, TearDown e SuiteTearDown. Não é fácil perceber essa duplicação, pois as quatro instâncias estão misturadas com outros códigos e não estão uniformemente repetidas. Mesmo assim, a duplicação é um problema, pois ela amontoa o código e serão necessárias quatro modificações se o algoritmo mudar. Além de serem quatro oportunidades para a omissão de um erro.

Sanou-se essa duplicação através do método `include` na Listagem 3.7. Leia este código novamente e note como a legibilidade do módulo inteiro foi melhorada com a retirada de tais repetições.

A duplicação pode ser a raiz de todo o mal no software. Muitos princípios e práticas têm sido criados com a finalidade de controlá-la ou eliminá-la. Considere, por exemplo, que todas as regras de normalização de bando de dados de Ted Codd servem para eliminar duplicação¹⁴ de dados. Considere também como a programação orientada a objeto serve para centralizar o código em classes-base que seriam outrora redundantes. Programação estruturada, Programação Orientada a Aspecto e Programação Orientada a Componentes são todas, em parte, estratégias para eliminar duplicação de código. Parece que desde a invenção da sub-rotina, inovações no desenvolvimento de software têm sido uma tentativa contínua para eliminar a duplicação de nossos códigos-fonte.



Programação estruturada

Alguns programadores seguem as regras programação estruturada de Edsger Dijkstra¹⁴, que disse que cada função e bloco dentro de uma função deve ter uma entrada e uma saída. Cumprir essas regras significa que deveria haver apenas uma instrução `return` na função, nenhum `break` ou `continue` num loop e jamais um `goto`.

Enquanto somos solidários com os objetivos e disciplinas da programação estruturada, tais regras oferecem pouca vantagem quando as funções são muito pequenas. Apenas em funções maiores tais regras proporcionam benefícios significativos.

Portanto, se você mantiver suas funções pequenas, então as várias instruções `return`, `break` ou `continue` casuais não trarão problemas e poderão ser até mesmo mais expressivas do que

¹¹. Aquelas que pensaram que poderiam se livrar da recompilação e da redistribuição foram encontrados, e tomadas as devidas providências.

a simples regra de uma entrada e uma saída. Por outro lado, o `goto` só faz sentido em funções grandes, portanto ele deve-se evitá-lo.

Como escrever funções como essa?

Criar um software é como qualquer outro tipo de escrita. Ao escrever um artigo, você primeiro coloca seus pensamentos no papel e depois os organiza de modo que fiquem fáceis de ler. O primeiro rascunho pode ficar desastroso e desorganizado, então você o apaga, reestrutura e refina até que ele fique como você deseja.

Quando escrevo funções, elas começam longas e complexas; há muitas endentações e loops aninhados; possuem longas listas de parâmetros; os nomes são arbitrários; e há duplicação de código. Mas eu também tenho uma coleção de testes de unidade que analisam cada uma dessas linhas desorganizadas do código.

Sendo assim, eu organizo e refino o código, divido funções, troco os nomes, elimino a duplicação, reduzo os métodos e os reorganizo. Às vezes, desmonto classes inteiras, tudo com os testes em execução.

No final, minhas funções seguem as regras que citei neste capítulo. Não as aplico desde o início. Acho que isso não seja possível.

Conclusão

Cada sistema é construído a partir de uma linguagem específica a um domínio desenvolvida por programadores para descrever o sistema. As funções são os verbos dessa linguagem, e classes os substantivos. Isso não é um tipo de retomada da antiga noção de que substantivos e verbos nos requerimentos de um documento sejam os primeiros palpites das classes e funções de um sistema. Mas sim uma verdade muito mais antiga. A arte de programar é, e sempre foi, a arte do projeto de linguagem.

Programadores experientes veem os sistemas como histórias a serem contadas em vez de programas a serem escritos. Eles usam os recursos da linguagem de programação que escolhem para construir uma linguagem muito mais rica e expressiva do que a usada para contar a estória. Parte da linguagem específica a um domínio é a hierarquia de funções que descreve todas as ações que ocorrem dentro daquele sistema. Em um ato engenhoso, escrevem-se essas funções para usar a mesma linguagem específica a um domínio que eles criaram para contar sua própria parte da história.

Este capítulo falou sobre a mecânica de se escrever bem funções. Se seguir as regras aqui descritas, suas funções serão curtas, bem nomeadas e bem organizadas. Mas jamais se esqueça de que seu objetivo verdadeiro é contar a história do sistema, e que as funções que você escrever precisam estar em perfeita sincronia e formar uma linguagem clara e precisa para lhe ajudar na narração.

SetupTeardownIncluder

Listagem 3-7

SetupTeardownIncluder.java

```
package fitnesse.html;

import fitnesse.responders.run.SuiteResponder;
import fitnesse.wiki.*;

public class SetupTeardownIncluder {
    private PageData pageData;
    private boolean isSuite;
    private WikiPage testPage;
    private StringBuffer newPageContent;
    private PageCrawler pageCrawler;

    public static String render(PageData pageData) throws Exception
        return render(pageData, false);
    }

    public static String render(PageData pageData, boolean isSuite)
        throws Exception {
        return new SetupTeardownIncluder(pageData).render(isSuite);
    }

    private SetupTeardownIncluder(PageData pageData) {
        this.pageData = pageData;
        testPage = pageData.getWikiPage();
        pageCrawler = testPage.getPageCrawler();
        newPageContent = new StringBuffer();
    }

    private String render(boolean isSuite) throws Exception {
        this.isSuite = isSuite;
        if (isTestPage())
            includeSetupAndTeardownPages();
        return pageData.getHtml();
    }

    private boolean isTestPage() throws Exception {
        return pageData.hasAttribute("Test");
    }

    private void includeSetupAndTeardownPages() throws Exception {
        includeSetupPages();
        includePageContent();
        includeTeardownPages();
        updatePageContent();
    }
}
```

Listagem 3-7 (continuação):**SetupTeardownIncluder.java**

```
private void includeSetupPages() throws Exception {
    if (isSuite)
        includeSuiteSetupPage();
    includeSetupPage();
}

private void includeSuiteSetupPage() throws Exception {
    include(SuiteResponder.SUITE_SETUP_NAME, "-setup");
}

private void includeSetupPage() throws Exception {
    include("SetUp", "-setup");
}

private void includePageContent() throws Exception {
    newPageContent.append(pageData.getContent());
}

private void includeTeardownPages() throws Exception {
    includeTeardownPage();
    if (isSuite)
        includeSuiteTeardownPage();
}

private void includeTeardownPage() throws Exception {
    include("TearDown", "-teardown");
}

private void includeSuiteTeardownPage() throws Exception {
    include(SuiteResponder.SUITE_TEARDOWN_NAME, "-teardown");
}

private void updatePageContent() throws Exception {
    pageData.setContent(newPageContent.toString());
}

private void include(String pageName, String arg) throws Exception {
    WikiPage inheritedPage = findInheritedPage(pageName);
    if (inheritedPage != null) {
        String pagePathName = getPathNameForPage(inheritedPage);
        buildIncludeDirective(pagePathName, arg);
    }
}

private WikiPage findInheritedPage(String pageName) throws Exception {
    return PageCrawlerImpl.getInheritedPage(pageName, testPage);
}

private String getPathNameForPage(WikiPage page) throws Exception {
    WikiPagePath pagePath = pageCrawler.getFullPath(page);
    return PathParser.render(pagePath);
}

private void buildIncludeDirective(String pagePathName, String arg) {
    newPageContent
        .append("\n!include ")
}
```

Comentários



"Não insira comentários num código ruim, reescreva-o".

—Brian W. Kernighan e P. J. Plaugher¹

Nada pode ser tão útil quanto um comentário bem colocado. Nada consegue amontoar um módulo mais do que comentários dogmáticos e supérfluos. Nada pode ser tão prejudicial quanto um velho comentário mal feito que dissemina mentiras e informações incorretas.

Comentários não são como a Lista de Schindler. Não são o “bom puro”. De fato, eles são, no máximo, um mal necessário. Se nossas linguagens de programação fossem expressivas o suficiente ou se tivéssemos o talento para manipular com destreza tais linguagens de modo a expressar nossa intenção, não precisaríamos de muitos comentários, quiçá nenhum.

O uso adequado de comentários é compensar nosso fracasso em nos expressar no código. Observe que usei a palavra fracasso. E é isso que eu quis dizer. Comentários são sempre fracassos. Devemos usá-los porque nem sempre encontramos uma forma de nos expressar sem eles, mas seu uso não é motivo para comemoração.

Então, quando você estiver numa situação na qual precise criar um comentário, pense bem e veja se não há como se expressar através do código em si. Toda vez que você fizer isso, dê em si mesmo um tapinha de aprovação nas costas. Toda vez que você escrever um comentário, faça uma careta e sinta o fracasso de sua capacidade de expressão.

Por que sou não gosto de comentários? Porque eles mentem. Nem sempre, e não intencionalmente, mas é muito comum. Quanto mais antigo um comentário for e quanto mais longe estiver do código o qual ele descreve, mais provável será que esteja errado. O motivo é simples. Não é realístico que programadores consigam mantê-los atualizados.

Códigos mudam e evoluem. Movem-se blocos para lá e para cá, que se bifurcam e se reproduzem e se unem novamente, formando monstros gigantescos. Infelizmente, os comentários nem sempre os seguem – nem sempre é possível. E, muito frequentemente, os comentários ficam longe do código o qual descrevem e se tornam dizeres órfãos com uma exatidão cada vez menor. Por exemplo, olhe o que aconteceu com o comentário abaixo e a linha que ele procurava descrever:

```
MockRequest request;
private final String HTTP_DATE_REGEX = 
    "[SMTWF][a-z]{2}\\,\\s[0-9]{2}\\s[JFMASOND][a-z]{2}\\s"+
    "[0-9]{4}\\s[0-9]{2}::[0-9]{2}::[0-9]{2}\\sGMT";
private Response response;
private FitNesseContext context;
private FileResponder responder;
private Locale saveLocale;
// Exemplo: "Tue, 02 Apr 2003 22:18:49 GMT"
```

Outras instâncias de variáveis que provavelmente foram adicionadas posteriormente ficaram entre a constante `HTTP_DATE_REGEX` e seu comentário descritivo.

É possível dizer que os programadores deveriam ser disciplinados o bastante para manter os comentários em um elevado estado de atualização, relevância e precisão. Concordo que deveriam. Mas eu preferiria que essa energia fosse direcionada para tornar o código tão claro e descritivo que de início nem se precisaria de comentários.

Comentários imprecisos são muito piores do que nenhum. Eles enganam e iludem; deixam expectativas que jamais serão cumpridas; citam regras antigas que não precisariam mais, ou não deveriam, ser seguidas.

Só se pode encontrar a verdade em um lugar: no código. Só ele pode realmente lhe dizer o que ele faz. Ele é a única fonte de informações verdadeiramente precisas. Entretanto, embora às vezes comentários sejam necessários, gastaríamos energia considerável para minimizá-los.

Comentários Compensam um Código Ruim

Uma das motivações mais comuns para criar comentários é um código ruim. Construímos um módulo e sabemos que está confuso e desorganizado. Estamos cientes da bagunça. Nós mesmos dizemos “Oh, é melhor inserir um comentário!”. Não! É melhor limpá-lo.

Códigos claros e expressivos com poucos comentários são de longe superiores a um amontoado e complexo com muitos comentários. Ao invés de gastar seu tempo criando comentários para explicar a bagunça que você fez, use-o para limpar essa zona.

Explique-se no código

Certamente há vezes em que não é possível se expressar direito no código. Infelizmente, devido a isso, muitos programadores assumiram que o código raramente é, se é que possa ser, um bom meio para se explicar. Evidentemente isso é falso. O que você preferiria ver? Isso:

```
// Verifica se o funcionario tem direito a todos os beneficios  
if ((employee.flags & HOURLY_FLAG) &&  
    (employee.age > 65))
```

Ou isso?

```
if (employee.isEligibleForFullBenefits())
```

Só é preciso alguns segundos de pensamento para explicar a maioria de sua intenção no código. Em muitos casos, é simplesmente uma questão de criar uma função cujo nome diga a mesma coisa que você deseja colocar no comentário.

Comentários Bons

Certos comentários são necessários ou benéficos. Veremos alguns que considero valerem os bits que consumem. Tenha em mente, contudo, que o único comentário verdadeiramente bom é aquele em que você encontrou uma forma para não escrevê-lo.

Comentários Legais

Às vezes, nossos padrões de programação corporativa nos forçam a escrever certos comentários por questões legais. Por exemplo, frases sobre direitos autorais e autoria são informações necessárias e lógicas para se colocar no início de um arquivo fonte.

Por exemplo, abaixo está o comentário padrão de cabeçalho que colocamos no início de todo arquivo fonte do FitNesse. Fico feliz em dizer que nossa IDE evita a união automática desse comentário para que não fique aglomerado.

```
// Direitos autorais (C) 2003,2004,2005 por Object Mentor, Inc. Todos  
os direitos reservados.  
// Distribuido sob os termos da versão 2 ou posterior da Licenca  
Publica Geral da GNU.
```

Comentários como esse não devem ser contratos ou termos legais. Onde for possível, faça referência a uma licença padrão ou outro documento externo em vez de colocar todos os termos e condições no mesmo comentário.

Comentários Informativos

Às vezes é prático fornecer informações básicas em um comentário. Por exemplo, considere o comentário abaixo que explica o valor retornado de um método abstrato:

```
// Retorna uma instancia do Responder sendo testado.
protected abstract Responder responderInstance();
```

Um comentário como este pode ser útil às vezes, mas, sempre que possível, é melhor usar o nome da função para transmitir a informação. Por exemplo, neste caso, o comentário ficaria redundante se trocássemos o nome da função: responderBeingTested.

Assim ficaria um pouco melhor:

```
// formato igual a kk:mm:ss EEE, MMM dd, aaaa
Pattern timeMatcher = Pattern.compile("\d*:\d*:\d* \w*, \w* \d*, \d*");
```

Neste caso, o comentário nos permite saber que a expressão regular deve combinar com uma hora e data formatadas com a função SimpleDateFormat.format usando a string específica com o formato. Mesmo assim, teria ficado melhor e mais claro se esse código tivesse sido colocado em uma classe especial para converter os formatos de datas e horas. Então, o comentário provavelmente seria supérfluo.

Explicação da intenção

Às vezes, um comentário vai além de ser apenas informações úteis sobre a implementação e fornece a intenção por trás de uma decisão. No caso a seguir, vemos uma decisão interessante documentada através de um comentário. Ao comparar dois objetos, o autor decidiu que queria classificar como superiores os objetos de sua classe em relação aos de outras.

```
public int compareTo(Object o)
{
    if(o instanceof WikiPagePath)
    {
        WikiPagePath p = (WikiPagePath) o;
        String compressedName = StringUtil.join(names, " ");
        String compressedArgumentName = StringUtil.join(p.names, " ");
        return compressedName.compareTo(compressedArgumentName);
    }
    return 1; // somos superiores porque somos tipo certo.
}
```

Abaixo está um exemplo melhor ainda. Talvez você discorde da solução do programador, mas pelo menos você sabe o que ele estava tentando fazer.

```
public void testConcurrentAddWidgets() throws Exception {
```

```
WidgetBuilder widgetBuilder =
    new WidgetBuilder(new Class[]{BoldWidget.class});
String text = "bold text";
```

```
ParentWidget parent =
    new BoldWidget(new MockWidgetRoot(), "bold text");
AtomicBoolean failFlag = new AtomicBoolean();
failFlag.set(false);

//Essa é a nossa melhor tentativa para conseguir uma condição de corrida.
//Para isso criamos um grande número de threads.
for (int i = 0; i < 25000; i++) {
    WidgetBuilderThread widgetBuilderThread =
        new WidgetBuilderThread(widgetBuilder, text, parent,
failFlag);
    Thread thread = new Thread(widgetBuilderThread);
    thread.start();
}
assertEquals(false, failFlag.get());
}
```

Esclarecimento

Às vezes é bom traduzir o significado de alguns parâmetros ou valores retornados obscuros para algo inteligível. De modo geral, é melhor encontrar uma forma de esclarecer tal parâmetro ou valor retornado por si só, mas quando for parte da biblioteca padrão, ou de um código que não se possa alterar, então um comentário esclarecedor pode ser útil.

```
public void testCompareTo() throws Exception
{
    WikiPagePath a = PathParser.parse("PageA");
    WikiPagePath ab = PathParser.parse("PageA.PageB");
    WikiPagePath b = PathParser.parse("PageB");
    WikiPagePath aa = PathParser.parse("PageA.PageA");
    WikiPagePath bb = PathParser.parse("PageB.PageB");
    WikiPagePath ba = PathParser.parse("PageB.PageA");
    assertTrue(a.compareTo(a) == 0); // a == a
    assertTrue(a.compareTo(b) != 0); // a != b
    assertTrue(ab.compareTo(ab) == 0); // ab == ab
    assertTrue(a.compareTo(b) == -1); // a < b
    assertTrue(aa.compareTo(ab) == -1); // aa < ab
    assertTrue(ba.compareTo(bb) == -1); // ba < bb
    assertTrue(b.compareTo(a) == 1); // b > a
    assertTrue(ab.compareTo(aa) == 1); // ab > aa
    assertTrue(bb.compareTo(ba) == 1); // bb > ba
}
```

Há um risco considerável, é claro, de que um comentário esclarecedor possa estar incorreto. Leia o exemplo anterior e veja como é difícil verificar se estão corretos. Isso explica tanto o porquê da necessidade do esclarecimento como seu risco. Portanto, antes de criar comentários como esses, certifique-se de que não há outra saída melhor e, então, certifique-se ainda mais se estão precisos.

Alerta Sobre Consequências

Às vezes é útil alertar outros programadores sobre certas consequências. Por exemplo, o comentário abaixo explica porque um caso de teste em particular está desabilitado:

```
// Não execute a menos que você
// tenha tempo disponível.

public void _testWithReallyBigFile()
{
    writeLinesToFile(10000000);
    response.setBody(testFile);
    response.readyToSend(this);
    String responseString = output.
toString();
    assertEquals("Content-Length: 1000000000", responseString);
    assertTrue(bytesSent > 1000000000);
}
```



Hoje em dia, desabilitariamos o teste de caso através do atributo `@ignore` com uma string explanatória adequada: `@ignore ("Leva muito tempo para executar")`. Antes da chegada do JUnit4, uma convenção comum era colocar um traço inferior (*underscore*) no início do nome do método.

O comentário, enquanto divertido, passa sua mensagem muito bem.

Outro exemplo mais direto seria:

```
public static SimpleDateFormat makeStandardHttpDateFormat()
{
    // SimpleDateFormat não é uma thread segura,
    // é preciso criar cada instância independentemente.
    SimpleDateFormat df = new SimpleDateFormat("EEE, dd MMM yyyy
HH:mm:ss z");
    df.setTimeZone(TimeZone.getTimeZone("GMT"));
    return df;
}
```

Talvez você reclame por haver melhores maneiras de resolver esse problema. Talvez eu concorde com você, mas o comentário como foi feito é perfeitamente lógico. Ele evitara que um programador afoito use um inicializador estático em prol da eficiência.

Comentário TODO

Às vezes é cabível deixar notas “To Do” (‘Fazer’) em comentários no formato `//TODO`. No caso a seguir, o comentário TODO explica por que a função tem uma implementação degradante e o que se deveria fazer com aquela função.

```
//TODO-MdM essas não são necessárias
// Esperamos que isso não esteja mais aqui quando verificarmos o modelo
protected VersionInfo makeVersion() throws Exception
{
    return null;
}
```

TODOS são tarefas que os programadores acham que devem ser efetuadas, mas, por alguma razão, não podem no momento. Pode ser um lembrete para excluir uma instrução desnecessária ou um apelo para que alguém olhe o problema. Ou um pedido para que alguém pense em um nome melhor ou um lembrete para fazer uma alteração que é dependente de um evento determinado. Seja qual for o TODO, ele não justifica deixar um código ruim no sistema.

Hoje em dia, a maioria das IDEs oferecem ferramentas e recursos para localizar todos os comentários TODO; portanto, não é provável que fiquem perdidos no código. Mesmo assim, você não deseja que seu código fique amontoado de TODOS, sendo assim, procure-os regularmente e elimine os que puder.

Destaque

Pode-se usar um comentário para destacar a importância de algo que talvez pareça irrelevante.

```
String listItemContent = match.group(3).trim();
// a função trim é muito importante. Ela remove os espaços
// iniciais que poderiam fazer com que o item fosse
// reconhecido como outra lista.
new ListItemWidget(this, listItemContent, this.level + 1);
return buildList(text.substring(match.end()));
```

Javadocs em APIs Públicas

Não há nada de tão prático e satisfatório como uma API pública em descrita. Os javadocs para a biblioteca Java padrão são um exemplo. No máximo, seria difícil escrever programas Java sem eles.

Se estiver criando uma API pública, então você certamente deveria escrever bons javadocs para ela. Mas tenha em mente os outros conselhos neste capítulo. Os javadocs podem ser tão enganadores, não-locais e desonestos como qualquer outro tipo de comentário.

Comentários Ruins

A maioria dos comentários cai nesta categoria. Geralmente eles são suportes ou desculpas para um código de baixa qualidade ou justificativas para a falta de decisões, amontoados como se o programador estivesse falando com si mesmo.

Murmúrio

Usar um comentário só porque você sente que deve ou porque o processo o requer é besteira. Se optar criar um comentário, então gaste o tempo necessário para fazê-lo bem. Por exemplo, a seguir está um caso que encontrei no FitNesse, no qual um comentário poderia ter sido útil. Entretanto, o autor estava com pressa ou não prestava muita atenção. Seu murmúrio foi deixado para trás como um enigma:

```
public void loadProperties()
{
    try
```

```

    {
        String propertiesPath =
            propertiesLocation + "/" + PROPERTIES_FILE;
        FileInputStream propertiesStream =
            new FileInputStream(propertiesPath);
        loadedProperties.load(propertiesStream);
    }
    catch(IOException e)
    {
        // Nenhum arquivo de propriedades significa que todos os padrões
        // estão carregados
    }
}

```

O que esse comentário no bloco `catch` significa? Claramente fazia sentido para o autor, mas o significado não foi muito bem transmitido. Aparentemente, se capturarmos (`catch`) uma `IOException`, significaria que não há arquivo de propriedades; e, neste caso, todos os padrões estão carregados. Mas quem carrega os padrões? Eles foram carregados antes da chamada ao `loadProperties.load`? Ou este capturou a exceção, carregou os padrões e, então, passou a exceção para que ignorássemos? Ou `loadProperties.load` carregou todos os padrões antes de tentar carregar o arquivo? Será que o autor estava limpando sua consciência por ter deixado o bloco do `catch` vazio? Ou – e essa possibilidade é assustadora – ele estava tentando dizer a si mesmo para voltar depois e escrever o código que carregaria os padrões?

Nosso único recurso é examinar o código em outras partes do sistema para descobrir o que está acontecendo. Qualquer comentário que lhe obrigue a analisar outro módulo em busca de um significado falhou em transmitir sua mensagem e não vale os bits que consume.

Comentários Redundantes

A Listagem 4.1 uma função simples com um comentário no cabeçalho que é completamente redundante. Provavelmente leva-se mais tempo para lê-lo do que o código em si.

Listagem 4-1 `waitForClose`

```

// Utility method that returns when this.closed is true. Throws an exception
// if the timeout is reached.
public synchronized void waitForClose(final long timeoutMillis)
throws Exception
{
    if(!closed)
    {
        wait(timeoutMillis);
        if(!closed)
            throw new Exception("MockResponseSender could not be closed");
    }
}

```

Qual o propósito desse comentário? Certamente não é mais informativo do que o código. Nem mesmo justifica o código ou oferece uma intenção ou raciocínio. É mais fácil ler o código apenas. De fato, ele é menos preciso do que o código e induz o leitor a aceitar tal falta de precisão.

em vez da interpretação verdadeira. É mais como um vendedor interesseiro de carros usados lhe garantindo que não é preciso olhar sob o capô.

Agora considere o grande número de javadocs inúteis e redundantes na Listagem 4.2 retirada do Tomcat. Esses comentários só servem para amontoar e encobrir o código. Eles não passam informação alguma. Para piorar, só lhe mostrei os primeiros, mas há muito mais neste módulo.

Listagem 4-2**ContainerBase.java (Tomcat)**

```
public abstract class ContainerBase
    implements Container, Lifecycle, Pipeline,
    MBeanRegistration, Serializable {

    /**
     * The processor delay for this component.
     */
    protected int backgroundProcessorDelay = -1;

    /**
     * The lifecycle event support for this component.
     */
    protected LifecycleSupport lifecycle =
        new LifecycleSupport(this);

    /**
     * The container event listeners for this Container.
     */
    protected ArrayList listeners = new ArrayList();

    /**
     * The Loader implementation with which this Container is
     * associated.
     */
    protected Loader loader = null;

    /**
     * The Logger implementation with which this Container is
     * associated.
     */
    protected Log logger = null;

    /**
     * Associated logger name.
     */
    protected String logName = null;

    /**
     * The Manager implementation with which this Container is
     * associated.
     */
    protected Manager manager = null;
```

Listagem 4-2 (continuação):**ContainerBase.java (Tomcat)**

```
/**  
 * The cluster with which this Container is associated.  
 */  
protected Cluster cluster = null;  
  
/**  
 * The human-readable name of this Container.  
 */  
protected String name = null;  
  
/**  
 * The parent Container to which this Container is a child.  
 */  
protected Container parent = null;  
  
/**  
 * The parent class loader to be configured when we install a  
 * Loader.  
 */  
protected ClassLoader parentClassLoader = null;  
  
/**  
 * The Pipeline object with which this Container is  
 * associated.  
 */  
protected Pipeline pipeline = new StandardPipeline(this);  
  
/**  
 * The Realm with which this Container is associated.  
 */  
protected Realm realm = null;  
  
/**  
 * The resources DirContext object with which this Container  
 * is associated.  
 */  
protected DirContext resources = null;
```

Comentários Enganadores

Às vezes, com todas as melhores das intenções, um programador faz uma afirmação não muito clara em seus comentários. Lembre-se também do redundante e enganador comentário que vimos na Listagem 4.1.

Como você descobriu que o comentário era enganador? O método não retornava quando `this.closed` virava `true` (`verdadeiro`), mas só se `this.closed` já fosse `true` (`verdadeiro`); caso contrário, ele esperava por um tempo limite e, então, lançava uma exceção se `this.closed` ainda não fosse `true` (`verdadeiro`).

Essa pequena desinformação, expressada em um comentário mais difícil de ler do que o código em si, poderia fazer com que outro programador despreocupadamente chamassem essa função esperando-a que retornasse assim que `this.closed` se tornasse `true` (`verdadeiro`). Esse pobre programador logo se veria efetuando uma depuração tentando descobrir o porquê da lentidão do código.

Comentários Imperativos

É basicamente tolo ter uma regra dizendo que toda função deva ter um Javadoc, ou toda variável um comentário. Estes podem se amontoar no código, disseminar mentiras e gerar confusão e desorganização.

Por exemplo, os javadocs exigidos para cada função levariam a abominações, como as da Listagem 4.3. Essa zona não acrescenta nada e só serve para ofuscar o código e abrir o caminho para mentiras e desinformações.

Listagem 4-3

```
/**  
 *  
 * @param title The title of the CD  
 * @param author The author of the CD  
 * @param tracks The number of tracks on the CD  
 * @param durationInMinutes The duration of the CD in minutes  
 */  
public void addCD(String title, String author,  
                  int tracks, int durationInMinutes) {  
    CD cd = new CD();  
    cd.title = title;  
    cd.author = author;  
    cd.tracks = tracks;  
    cd.duration = duration;  
    cdList.add(cd);  
}
```

Comentários Longos

Às vezes, as pessoas, toda vez que editam um módulo, sempre adicionam um comentário no início. Após várias alterações, a quantidade de comentários acumulada parece mais uma redação ou um diário. Já vi módulos com dezenas de páginas assim.

```

* Changes (from 11-Oct-2001)
* -----
* 11-Oct-2001 : Re-organised the class and moved it to new package
*                 com.jrefinery.date (DG);
* 05-Nov-2001 : Added a getDescription() method, and eliminated NotableDate
*                 class (DG);
* 12-Nov-2001 : IBD requires setDescription() method, now that NotableDate
*                 class is gone (DG); Changed getPreviousDayOfWeek(),
*                 getFollowingDayOfWeek() and getNearestDayOfWeek() to correct
*                 bugs (DG);
* 05-Dec-2001 : Fixed bug in SpreadsheetDate class (DG);
* 29-May-2002 : Moved the month constants into a separate interface
*                 (MonthConstants) (DG);
* 27-Aug-2002 : Fixed bug in addMonths() method, thanks to N???levka Petr (DG);
* 03-Oct-2002 : Fixed errors reported by Checkstyle (DG);
* 13-Mar-2003 : Implemented Serializable (DG);
* 29-May-2003 : Fixed bug in addMonths method (DG);
* 04-Sep-2003 : Implemented Comparable. Updated the isInRange javadocs (DG);
* * 05-Jan-2005 : Fixed bug in addYears() method (1096282) (DG);

```

Há muito tempo, havia um bom motivo para criar e preservar essas informações no início de cada módulo, pois não existiam ainda os sistemas de controle de código fonte para fazer isso por nós. Hoje em dia, entretanto, esses comentários extensos são apenas entulhos que confundem o código, devendo assim ser completamente removidos.

Comentários Ruidosos

Às vezes você vê comentários que nada são além de “chiados”. Eles dizem o óbvio e não fornecem novas informações.

```

/**
 * Construtor padrão.
 */
protected AnnualDateRule() {
}

```

Ah, sério? Ou este:

```

/** Dia do mes. */
private int dayOfMonth;

```

E há também o seguinte tipo de redundância:

```

/**
 * Retorna o dia do mês.
 *
 * @return o dia do mês.
 */
public int getDayOfMonth() {
    return dayOfMonth;
}

```

Esses comentários são tão irrelevantes que aprendemos a ignorá-los. Ao termos o código, nossos olhos passam direto por eles. No final, os comentários passam a “mentir” conforme o código muda.

O primeiro comentário na Listagem 4.4 parece adequado². Ele explica por que o bloco catch é ignorado. Contudo, o segundo não passa de um chiado. Aparentemente, o programador estava tão frustrado por criar blocos try/catch na função que ele precisou desabafar.

Listagem 4-4 startSending

```
private void startSending()
{
    try
    {
        doSending();
    }
    catch(SocketException e)
    {
        // normal. someone stopped the request.
    }
    catch(Exception e)
    {
        try
        {
            response.add(ErrorResponder.makeExceptionString(e));
            response.closeAll();
        }
        catch(Exception e1)
        {
            //Give me a break!
        }
    }
}
```

Em vez de desabafar em comentários sem sentido e ruidosos, ele poderia ter pegado tal frustração e usado-a para melhorar a estrutura do código. Ele deveria ter redirecionado sua energia para colocar o último bloco try/catch em uma função separada, como mostra a Listagem 4.5.

Listagem 4-5 startSending (refatorado)

```
private void startSending()
{
    try
    {
        doSending();
    }
    catch(SocketException e)
    {
        // normal. someone stopped the request.
    }
    catch(Exception e)
    {
```

Listagem 4-5 (continuação)**startSending (refatorado)**

```

        addExceptionAndCloseResponse(e);
    }
}

private void addExceptionAndCloseResponse(Exception e)
{
    try
    {
        response.add(ErrorResponder.makeExceptionString(e));
        response.closeAll();
    }
    catch(Exception e1)
    {
    }
}
}

```

Troque a tentação para criar ruídos pela determinação para limpar seu código. Você perceberá que isso lhe tornará um programador melhor e mais feliz.

Ruídos assustadores

Os javadocs também podem ser vistos como ruídos. Qual o objetivo dos Javadoc (de uma biblioteca de código livre bem conhecida) abaixo?

```

/** Nome. */
private String name;

/** The version. */
private String version;

/** Nome da licença. */
private String licenceName;

/** Versão. */
private String info;

```

Releia os comentários com mais atenção. Notou um erro de recortar-colar? Se os autores não prestarem atenção na hora de escrever os comentários (ou colá-los), por que os leitores deveriam esperar algo de importante deles?

Evite o comentário se é possível usar uma função ou uma variável

Considere o pedaço de código abaixo:

```

// o módulo da lista global <mod> depende do
// subsistema do qual fazemos parte?
if (smodule.getDependSubsystems().contains(subSysMod.getSubSystem()))

```

Poderia-se evitar o comentário e usar:

```
ArrayList moduleDependees = smodule.getDependSubsystems();
String ourSubSystem = subSysMod.getSubSystem();
if (moduleDependees.contains(ourSubSystem))
```

O autor do código original talvez tenha escrito primeiro o comentário (pouco provável) e, então, o código de modo a satisfazer o comentário. Entretanto, o autor deveria ter refatorado o código, como eu fiz, para que pudesse remover o comentário.

Marcadores de Posição

Algumas vezes os programadores gostam de marcar uma posição determinada no arquivo fonte. Por exemplo, recentemente encontrei o seguinte num programa:

```
// Ações ////////////////////
```

É raro, mas há vezes que faz sentido juntar certas funções sob um indicador como esses. Mas, de modo geral, eles são aglomerações e devem-se excluí-los—especialmente as várias barras no final.

Pense assim: um indicador é chamativo e óbvio se você não os vê muito frequentemente. Portanto, use-os esporadicamente, e só quando gerarem benefícios significativos. Se usar indicadores excessivamente, eles cairão na categoria de ruidos e serão ignorados.

Comentários ao lado de chaves de fechamento

Às vezes, os programadores colocam comentários especiais ao lado de chaves de fechamento, como na Listagem 4.6.

Embora isso possa fazer sentido em funções longas com estruturas muito aninhadas, só serve para amontoar o tipo de funções pequenas e encapsuladas que preferimos. Portanto, se perceber uma vontade de comentar ao lado de chaves de fechamento, tente primeiro reduzir suas funções.

Listagem 4-6

wc.java

```
public class wc {
    public static void main(String[] args) {
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        String line;
        int lineCount = 0;
        int charCount = 0;
        int wordCount = 0;
        try {

            while ((line = in.readLine()) != null) {
                lineCount++;
                charCount += line.length();
                String words[] = line.split("\\W");
                wordCount += words.length;
            } //while
            System.out.println("wordCount = " + wordCount);
            System.out.println("lineCount = " + lineCount);
            System.out.println("charCount = " + charCount);
        } // try
    }
}
```

Listagem 4-6 (continuação)**wc.java**

```
    catch (IOException e) {
        System.err.println("Error:" + e.getMessage());
    } //catch
} //main
}
```

Créditos e autoria

/ Adicionado por Rick */*

Os sistemas de controle de código fonte são muito bons para lembrar que adicionou o quê e quando. Não há necessidade de poluir o código com comentários de autoria. Talvez você ache que tais comentários sejam úteis para ajudar outras pessoas a saberem o que falar sobre o código. Mas a verdade é que eles permanecem por anos, ficando cada vez menos precisos e relevantes.

Novamente, o sistema de controle de código fonte é um local melhor para este tipo de informação.

Explicação do código em comentários

Poucas práticas são tão condenáveis quanto explicar o código nos comentários. Não faça isso!

```
InputStreamResponse response = new InputStreamResponse();
response.setBody(formatter.getResultStream(), formatter.
getByteCount());
// InputStream resultsStream = formatter.getResultStream();
// StreamReader reader = new StreamReader(resultsStream);
// response.setContent(reader.read(formatter.getByteCount()));
```

Outros que vissem esse código não teriam coragem de excluir os comentários. Eles achariam que estão lá por um motivo e são importantes demais para serem apagados. Portanto, explicação de códigos em comentários se acumula como sujeira no fundo de uma garrafa de vinho ruim.

Observe o código do Commons do Apache:

```
this.bytePos = writeBytes(pngIdBytes, 0);
//hdrPos = bytePos;
writeHeader();
writeResolution();
//dataPos = bytePos;
if (writeImageData()) {
    writeEnd();
    this.pngBytes = resizeByteArray(this.pngBytes, this.maxPos);
}
else {
    this.pngBytes = null;
}
return this.pngBytes;
```

Por que aquelas duas linhas de código estão comentadas? Elas são importantes? Foram deixados como lembretes para alguma alteração iminente? Ou são apenas aglomerados que alguém comentara anos atrás e simplesmente não se preocupou em limpar?

Houve uma época, na década de 1960, em que explicar o código em comentários poderia ser prático. Mas já faz um tempo que temos os sistemas de controle de código fonte, que lembrarão o código para nós. Não precisamos explicá-lo em comentários. Simplesmente exclua o código. Prometo que não o perderá.

Comentários HTML

Códigos HTML em comentários de código fonte são uma aberração, como pode ver no código abaixo. Eles dificultam a leitura dos comentários onde seriam fáceis de ler – no editor/IDE. Se forem extrair comentários com alguma ferramenta (como o Javadoc) para exibir numa página da Web, então deveria ser responsabilidade de tal ferramenta, e não do programador, adicionar os comentários com os códigos HTML adequados.

```
/**  
 * Tarefa para executar os testes do Fit.  
 * Essa tarefa efetua os testes do FitNesse e exibe os resultados.  
 * <p/>  
 * <pre>  
 * Uso:  
 * <taskdef name="execute-fitnessse-tests"  
 * classname="fitnessse.ant.ExecuteFitnessseTestsTask";  
 * classpathref="classpath" />  
 * OU  
 * <taskdef classpathref="classpath"  
 * resource="tasks.properties" />  
 * <p/>  
 * <execute-fitnessse-tests  
 * suitepage="FitNesse.SuiteAcceptanceTests";  
 * fitnessseport="8082";  
 * resultsdir="${results.dir}";  
 * resultshtmlpage="fit-results.html";  
 * classpathref="classpath" />  
 * </pre>  
 */
```

Informações não-locais

Se você precisar escrever um comentário, então, coloque-o perto do código que ele descreve. Não forneça informações gerais do sistema no contexto de um comentário local. Considere, por exemplo, o comentário do Javadoc abaixo. Além de ser terrivelmente redundante, ele também fala sobre a porta padrão. Ainda assim, a função nem sabe que porta é essa. O comentário não está descrevendo a função, mas alguma outra em alguma parte distante do sistema. Certamente não há garantia de que esse comentário será atualizado quando o código que contém tal padrão for alterado.

```
/**  
 * Porta na qual o FitNesse deveria rodar. Padrão para <b>8082</b>.   
 * @param fitnesssePort  
 */  
public void setFitnesssePort(int fitnesssePort)  
{  
    this.fitnesssePort = fitnesssePort;  
}
```

Informações excessivas

Não adicione discussões históricas interessantes ou descrições irrelevantes de detalhes em seus comentários. Abaixo está o comentário de um módulo projetado para testar se uma função poderia codificar e decodificar base64. Além do número RFC, a pessoa que ler este código não precisa das informações históricas contidas no comentário.

```
/*
RFC 2045 - Multipurpose Internet Mail Extensions (MIME)
Parte um: seção 6.8 do Formato dos Corpos de Mensagens da Internet.
Codificação e Transferencia de Conteúdo em Base64
O processo de codificação representa grupos de entrada de 24 bits
como strings de saída de 4 caracteres codificados. Da esquerda para
a direita, forma-se um grupo de entrada de 24 bits pela concatenação
de 3 grupos de entrada de 8 bits.
Esses 24 bits serão, então, tratados como 4 grupos concatenados
de 6 bits, cada um traduzido para um único digito do alfabeto da
base64. Ao codificar um fluxo de bits através da codificação para
base64, deve-se presumir que tal fluxo esteja ordenado com o bit mais
significante vindo primeiro.
Ou seja, o primeiro bit no fluxo será o mais relevante no primeiro
byte de
8 bits, o oitavo será o bit menos relevante no mesmo primeiro byte
de 8 bits,
e assim por diante.
*/
```

Conexões nada óbvias

A conexão entre um comentário e o código que ele descreve deve ser óvia. Se for fazer um comentário, então você deseja, pelo menos, que o leitor seja capaz de ler o comentário e o código e, então, entender o que foi falado.

Considere, por exemplo, o comentário abaixo do Commons do Apache:

```
/*
 * começa com um array grande o bastante para conter todos os
pixels
 * (mais os bytes de filtragem) e 200 bytes extras para informações
no cabeçalho
*/
this.pngBytes = new byte[((this.width + 1) * this.height * 3) + 200];
```

O que é um byte de filtragem? Ele tem a ver com o +1? Ou com o *3? Com ambos? Um pixel é um byte? Por que 200? O objetivo de um comentário é explicar o que o código não consegue por si só. É uma lástima quando um comentário também precisa ser explicado.

Cabeçalhos de funções

Funções curtas não requerem muita explicação. Um nome bem selecionado para uma função pequena que faça apenas uma coisa costuma ser melhor do que um comentário no cabeçalho.

Javadocs em códigos não-públicos

Assim como os Javadocs são práticos para as APIs públicas, eles são uma maldição para o código não voltado para a distribuição ao público. Gerar páginas Javadoc para classes e funções dentro de um sistema geralmente não é prático, e a formalidade extra dos comentários javadocs unem um pouco mais de entulhos e distração.

Exemplo

Na Listagem 4.7, criei um módulo para o primeiro XP *Immersion* para servir de exemplo de má programação e estilo de comentário. Então, Kent Beck refatorou esse código para uma forma muito mais agradável na presença de algumas dezenas de estudantes empolgados. Mais tarde, adaptei o exemplo para meu livro *Agile Software Development, Principles, Patterns, and Practices* e o primeiro de meus artigos da coluna *Craftsman* publicados na revista *Software Development*.

Para mim, o fascinante desse módulo é que havia uma época quando muitos de nós o teríamos considerado “bem documentado”. Agora o vemos como uma pequena bagunça. Veja quantos problemas diferentes você consegue encontrar.

Listagem 4-7

GeneratePrimes.java

```
/**  
 * This class Generates prime numbers up to a user specified  
 * maximum. The algorithm used is the Sieve of Eratosthenes.  
 * <p>  
 * Eratosthenes of Cyrene, b. c. 276 BC, Cyrene, Libya --  
 * d. c. 194, Alexandria. The first man to calculate the  
 * circumference of the Earth. Also known for working on  
 * calendars with leap years and ran the library at Alexandria.  
 * <p>  
 * The algorithm is quite simple. Given an array of integers  
 * starting at 2. Cross out all multiples of 2. Find the next  
 * uncrossed integer, and cross out all of its multiples.  
 * Repeat until you have passed the square root of the maximum  
 * value.  
 *  
 * @author Alphonse  
 * @version 13 Feb 2002 atp  
 */  
import java.util.*;  
  
public class GeneratePrimes  
{  
    /**  
     * @param maxValue is the generation limit.  
     */  
    public static int[] generatePrimes(int maxValue)  
    {  
        if (maxValue >= 2) // the only valid case  
        {  
            // declarations  
            int s = maxValue + 1; // size of array  
            boolean[] f = new boolean[s];  
            int i;
```

Listagem 4-7 (continuação)

GeneratePrimes.java

```

// initialize array to true.
for (i = 0; i < s; i++)
    f[i] = true;

// get rid of known non-primes
f[0] = f[1] = false;

// sieve
int j;
for (i = 2; i < Math.sqrt(s) + 1; i++)
{
    if (f[i]) // if i is uncrossed, cross its multiples
    {
        for (j = 2 * i; j < s; j += i)
            f[j] = false; // multiple is not prime
    }
}

// how many primes are there?
int count = 0;
for (i = 0; i < s; i++)
{
    if (f[i])
        count++; // bump count.
}

int[] primes = new int[count];

// move the primes into the result
for (i = 0, j = 0; i < s; i++)
{
    if (f[i]) // if prime
        primes[j++] = i;
}

return primes; // return the primes
}
else // maxValue < 2
    return new int[0]; // return null array if bad input.
}

```

Na Listagem 4.8 pode-se ver uma versão refatorada do mesmo módulo. Note que o uso de comentários foi limitado de forma significativa – há apenas dois no módulo inteiro, e ambos são auto-explicativos.

Listagem 4.8:

PrimeGenerator.java (refatorado)

```
/**  
 * This class Generates prime numbers up to a user specified  
 * maximum. The algorithm used is the Sieve of Eratosthenes  
 * Given an array of integers starting at 2:  
 * Find the first uncrossed integer, and cross out all its
```

Listagem 4-8 (continuação)

PrimeGenerator.java (refatorado)

```
* multiples. Repeat until there are no more multiples
* in the array.
*/
public class PrimeGenerator
{
    private static boolean[] crossedOut;
    private static int[] result;

    public static int[] generatePrimes(int maxValue)
    {
        if (maxValue < 2)
            return new int[0];
        else
        {
            uncrossIntegersUpTo(maxValue);
            crossOutMultiples();
            putUncrossedIntegersIntoResult();
            return result;
        }
    }

    private static void uncrossIntegersUpTo(int maxValue)
    {
        crossedOut = new boolean[maxValue + 1];
        for (int i = 2; i < crossedOut.length; i++)
            crossedOut[i] = false;
    }

    private static void crossOutMultiples()
    {
        int limit = determineIterationLimit();
        for (int i = 2; i <= limit; i++)
            if (notCrossed(i))
                crossOutMultiplesOf(i);
    }

    private static int determineIterationLimit()
    {
        // Every multiple in the array has a prime factor that
        // is less than or equal to the root of the array size,
        // so we don't have to cross out multiples of numbers
        // larger than that root.
        double iterationLimit = Math.sqrt(crossedOut.length);
        return (int) iterationLimit;
    }

    private static void crossOutMultiplesOf(int i)
    {
        for (int multiple = 2*i;
             multiple < crossedOut.length;
             multiple += i)
            crossedOut[multiple] = true;
    }
}
```

Listagem 4-8 (continuação)

PrimeGenerator.java (refatorado)

```
private static boolean notCrossed(int i)
{
    return crossedOut[i] == false;
}

private static void putUncrossedIntegersIntoResult()
{
    result = new int[numberOfUncrossedIntegers()];
    for (int j = 0, i = 2; i < crossedOut.length; i++)
        if (notCrossed(i))
            result[j++] = i;
}

private static int numberOfUncrossedIntegers()
{
    int count = 0;
    for (int i = 2; i < crossedOut.length; i++)
        if (notCrossed(i))
            count++;

    return count;
}
```

Como o primeiro comentário é muito parecido com a função generatePrimes, fica fácil dizer que ele é redundante. Mesmo assim, acho que o comentário serve para facilitar a leitura do algoritmo, portanto prefiro mantê-lo.

Já o segundo se faz praticamente necessário. Ele explica a lógica por trás do uso da raiz quadrada como o limite da iteração. Não consegui encontrar um nome simples para a variável ou qualquer estrutura diferente de programação que esclarecesse esse ponto. Por outro lado, o uso da raiz quadrada poderia ser um conceito. Realmente estou economizando tanto tempo assim ao limitar a iteração à raiz quadrada? O cálculo desta demora mais do que o tempo que economizo?

Vale a pena ponderar. Usar a raiz quadrada como o limite da iteração satisfaz o hacker em mim que usa a antiga linguagem C e Assembly, mas não estou convencido de que compense o tempo e o esforço que todos gastariam para entendê-la.

Bibliografia

[KP78]: Kernighan and Plaugher, *The Elements of Programming Style*, 2d. ed., McGraw-Hill, 1978.

Então, quando estamos criando um software, devemos pensar que o resultado final é só o resultado final, mas é o resultado final que é o resultado final. Devemos pensar que o resultado final é só o resultado final, mas é o resultado final que é o resultado final.

Introdução à programação

5

Quando programamos, o que fazemos é programar o resultado final. Mas é o resultado final que é o resultado final.

É isso que fazemos quando programamos. Quando programamos, o que fazemos é programar o resultado final. Mas é o resultado final que é o resultado final.

Formatação



Quando as pessoas olham o código, desejamos que fiquem impressionadas com a polidez, a consistência e a atenção aos detalhes presentes. Queremos que reparem na organização. Desejamos que suas sobrancelhas se levantem ao percorrerem os módulos; que percebam que foram profissionais que estiveram ali. Se, em vez disso, virem um emaranhado de código como

se tivesse sido escrito por um bando de marinheiros bêbados, então provavelmente concluirão que essa mesma falta de atenção foi perpetuada por todo o projeto.

Você deve tomar conta para que seu código fique bem formatado, escolher uma série de regras simples que governem seu código e, então, aplicá-la de forma consistente. Se estiver trabalhando em equipe, então, todos devem concordar com uma única série de regras de formatação. Seria bom ter uma ferramenta automatizada que possa aplicar essas regras para você.

O objetivo da formatação

Primeiro de tudo, sejamos claros. A formatação do código é *importante*. Importante demais para se ignorar e importante demais para ser tratada religiosamente. Ela serve como uma comunicação, e essa é a primeira regra nos negócios de um desenvolvedor profissional.

Talvez você pensasse que “fazer funcionar” fosse a primeira regra. Espero, contudo, que, a esta altura, este livro já tenha tirado esse conceito de sua mente. A funcionalidade que você cria hoje tem grandes chances de ser modificada na próxima distribuição, mas a legibilidade de seu código terá um grande efeito em todas as mudanças que serão feitas. A formatação do código e a legibilidade anteriores que continuam a afetar a capacidade de extensão e de manutenção tempos após o código original foram alteradas além de reconhecimento. Seu estilo e disciplina sobrevivem, mesmo que seu código não.

Então quais as questões sobre formatação que nos ajuda a comunicar melhor?

Formatação vertical

Comecemos com o tamanho vertical. O seu código-fonte deve ser de que tamanho? Em Java, o tamanho do arquivo está intimamente relacionado ao da classe. Discutiremos sobre o tamanho das classes quando falarmos sobre classes. Mas, por agora, consideremos apenas o tamanho do arquivo.

Qual o tamanho da maioria dos códigos-fonte em Java? Há uma grande diversidade de tamanhos e algumas diferenças notáveis em estilo (veja a Figura 5.1). Há sete projetos diferentes na figura: Junit, FitNesse, testNG, Time and Money, JDepend, Ant e Tomcat. As linhas verticais mostram os comprimentos mínimo e máximo em cada projeto. A caixa exibe aproximadamente um terço (um desvio padrão¹) dos arquivos. O meio da caixa é a média. Portanto, o tamanho médio do código no projeto FitNesse é de cerca de 65 linhas, e cerca de um terço dos arquivos estão entre 40 e 100+ linhas. O maior arquivo no FitNesse tem aproximadamente 400 linhas, e o menor 6.

Note que essa é uma escala logarítmica; portanto, a pequena diferença na posição vertical indica uma diferença muito grande para o tamanho absoluto.

Junit, FitNesse e Time and Money são compostos de arquivos relativamente pequenos. Nenhum ultrapassa 500 linhas e a maioria dos arquivos tem menos de 200 linhas. Tomcat e Ant, por outro lado, têm alguns arquivos com milhares de linhas e outros, próximos à metade, ultrapassam 200 linhas.

O que isso nos diz? Parece ser possível construir sistemas significativos (o FitNesse tem quase 50.000 linhas) a partir de códigos simples de 200 linhas, com um limite máximo de 500. Embora essa não deva ser uma regra fixa, deve-se considerá-la bastante, pois arquivos pequenos costumam ser mais fáceis de se entender do que os grandes.

¹ A caixa mostra $\sigma/2$ acima e abaixo da média. Isso, sei que a distribuição do comprimento do arquivo não é normal, e, portanto, o desvio padrão não é

A metáfora do jornal

Pense num artigo de jornal bem redigido. Você o lê verticalmente. No topo você espera ver uma manchete que lhe diz do que se trata a estória e lhe permite decidir se deseja ou não ler. O primeiro parágrafo apresenta uma sinopse da estória toda, omitindo todos os detalhes, falando de uma maneira mais geral. Ao prosseguir a leitura, verticalmente, vão surgindo mais detalhes até que datas, nomes, citações, alegações e outras minúcias sejam apresentadas.

Desejamos que um código fonte seja como um artigo de jornal. O nome deve ser simples mas descritivo. O nome em si deve ser o suficiente para nos dizer se estamos no módulo certo ou não. As partes mais superiores do código-fonte devem oferecer os conceitos e algoritmos de alto nível. Os detalhes devem ir surgindo conforme se move para baixo, até encontrarmos os detalhes e as funções de baixo nível no código-fonte.

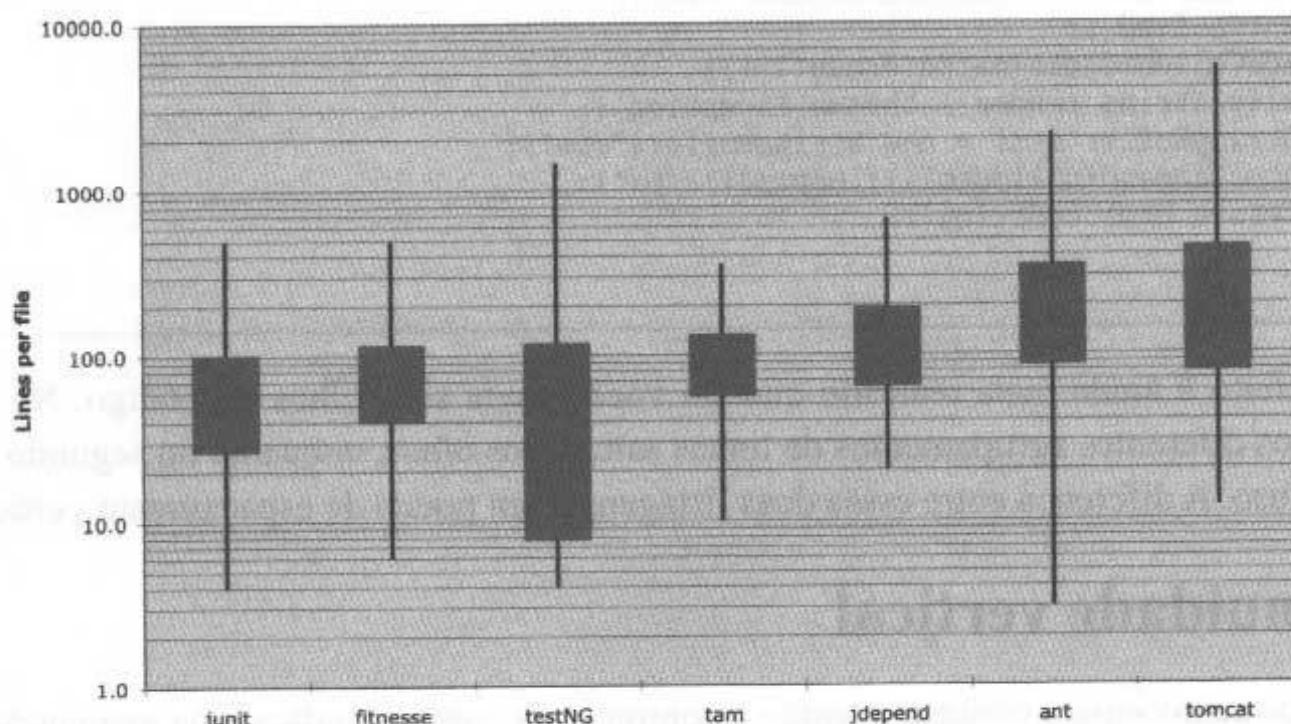


Figura 5.1:

Escala logarítmica de distribuição de tamanho de arquivos (altura da caixa = sigma)

Um jornal é composto de muitos artigos; a maioria é bastante pequena. Alguns são um pouco maiores. Muito poucos possuem textos que preencham a página toda. Isso torna o jornal *aproveitável*. Se ele fosse apenas uma estória extensa com uma aglomeração desorganizada de fatos, datas e nomes, nós simplesmente não o leríamos.

Espaçamento vertical entre conceitos

Quase todo código é lido da esquerda para a direita e de cima para baixo. Cada linha representa uma expressão ou uma estrutura, e cada grupo de linhas representa um pensamento completo. Esses pensamentos devem ficar separados por linhas em branco.

Considere, por exemplo, a Listagem 5.1. Há linhas em branco que separam a declaração e a importação do pacote e cada uma das funções. Essa simples e extrema regra tem grande impacto no layout visual do código. Cada linha em branco indica visualmente a separação entre conceitos. Ao descer pelo código, seus olhos param na primeira linha após uma em branco.

Retirar essas linhas em branco, como na Listagem 5.2, gera um efeito notavelmente obscuro na legibilidade do código.

Listagem 5-2

BoldWidget.java

```
package fitnessse.wikitext.widgets;
import java.util.regex.*;
public class BoldWidget extends ParentWidget {
    public static final String REGEXP = "'''.+?'''";
    private static final Pattern pattern = Pattern.compile("'''(.+?)'''",
        Pattern.MULTILINE + Pattern.DOTALL);
    public BoldWidget(ParentWidget parent, String text) throws Exception {
        super(parent);
        Matcher match = pattern.matcher(text);
        match.find();
        addChildWidgets(match.group(1));
    }
    public String render() throws Exception {
        StringBuffer html = new StringBuffer("<b>");
        html.append(childHtml()).append("</b>");
        return html.toString();
    }
}
```

Esse efeito é ainda mais realçado quando você desvia seus olhos do código. No primeiro exemplo, os diferentes agrupamentos de linhas saltam aos olhos, enquanto no segundo tudo fica meio confuso. A diferença entre essas duas listagens é um pouco de espaçamento vertical.

Continuidade vertical

Se o espaçamento separa conceitos, então a continuidade vertical indica uma associação íntima. Assim, linhas de código que estão intimamente relacionadas devem aparecer verticalmente unidas. Note como os comentários inúteis na Listagem 5.3 quebram essa intimidade entre a instância de duas variáveis.

Listagem 5.3

```
public class ReporterConfig {

    /**
     * The class name of the reporter listener
     */
    private String m_className;

    /**
     * The properties of the reporter listener
     */
    private List<Property> m_properties = new ArrayList<Property>();

    public void addProperty(Property property) {
        m_properties.add(property);
    }
}
```

A Listagem 5.4 está muito mais fácil de se ler. Ela cabe numa única visão, pelo menos para mim. Posso olhá-la e ver que é uma classe com duas variáveis e um método, sem ter de mover muito minha cabeça ou meus olhos. A listagem anterior me faz usar mais o movimento dos olhos e da cabeça para obter o mesmo nível de entendimento.

Listagem 5.4

```
public class ReporterConfig {  
    private String m_className;  
    private List<Property> m_properties = new ArrayList<Property>();  
  
    public void addProperty(Property property) {  
        m_properties.add(property);  
    }
```

Distância vertical

Já ficou tentando se encontrar numa classe, passando de uma função para a próxima, subindo e descendo pelo código-fonte, tentando adivinhar como as funções se relacionam e operam, só para se perder nesse labirinto de confusão? Já subiu pela estrutura de herança buscando a definição de uma variável ou função? Isso é frustrante, pois você está tentando entender *o que* o sistema faz, enquanto gasta tempo e energia mental numa tentativa de localizar e lembra *onde* estão as peças.

Os conceitos intimamente relacionados devem ficar juntos verticalmente [G10].

Obviamente essa regra não funciona para conceitos em arquivos separados. Mas, então, não se devem separar em arquivos distintos conceitos intimamente relacionados, a menos que tenha uma razão muito boa. Na verdade, esse é um dos motivos por que se devem evitar variáveis protegidas. Para os conceitos que são tão intimamente relacionados e que estão no mesmo arquivo-fonte, a separação vertical deles deve ser uma medida do quanto importante eles são para a inteligibilidade um do outro. Queremos evitar que nossos leitores tenham de ficar visualizando vários dos nossos arquivos-fonte e classes.

Declaração de variáveis. Devem-se declarar as variáveis o mais próximo possível de onde serão usadas.

Como nossas funções são muito pequenas, as variáveis locais devem ficar no topo de cada função, como mostra a função razoavelmente longa abaixo do Junit4.3.1.

```
private static void readPreferences() {  
    InputStream is= null;  
    try {  
        is= new FileInputStream(getPreferencesFile());  
        setPreferences(new Properties(getPreferences()));  
        getPreferences().load(is);  
    } catch (IOException e) {  
        try {  
            if (is != null)  
                is.close();  
        } catch (IOException e1) {  
        }  
    }  
}
```

Geralmente, devem-se declarar as variáveis de controle para loops dentro da estrutura de iteração, como mostra essa pequenina função da mesma fonte acima.

```
public int countTestCases() {  
    int count= 0;  
    for (Test each : tests)  
        count += each.countTestCases();  
    return count;  
}
```

Em raros casos pode-se declarar uma variável no início de um bloco ou logo depois de um loop em uma função razoavelmente longa. Veja um exemplo no pedacinho abaixo de uma função muito extensa do TestNG.

```
...  
for (XmlTest test : m_suite.getTests()) {  
    TestRunner tr = m_runnerFactory.newTestRunner(this, test);  
    tr.addListener(m_textReporter);  
    m_testRunners.add(tr);  
  
    invoker = tr.getInvoker();  
  
    for (ITestNGMethod m : tr.getBeforeSuiteMethods()) {  
        beforeSuiteMethods.put(m.getMethod(), m);  
    }  
  
    for (ITestNGMethod m : tr.getAfterSuiteMethods()) {  
        afterSuiteMethods.put(m.getMethod(), m);  
    }  
}  
...
```

Instâncias de variáveis. Por outro lado, devem-se declarar as instâncias de variáveis no início da classe. Isso não deve aumentar a distância vertical entre tais variáveis, pois, numa classe bem projetada, elas são usadas por muitos, senão todos, os métodos da classe.

Muito já se discutiu sobre onde devem ficar as instâncias de variáveis. Em C++ é comum a *regra da tesoura*, na qual colocamos todas as instâncias das variáveis no final. Em java, contudo, a convenção é colocá-las no início da classe.

Não vejo motivo para seguir uma ou outra convenção. O importante é que as instâncias de variáveis sejam declaradas em um local bem conhecido. Todos devem saber onde buscar as declarações.

Considere, por exemplo, o estranho caso da classe `TestSuite` no JUnit 4.3.1. Resumi bastante essa classe para mostrar a questão. Se você ler até cerca de metade do código, verá duas instâncias de variáveis declaradas. Seria difícil ocultá-las num lugar melhor. Quem ler este código se depararia por acaso com as declarações (como ocorreu comigo).

```
public class TestSuite implements Test {  
    static public Test createTest(Class<? extends TestCase>
```

```
theClass,  
  
String name) {  
    ...  
}  
  
public static Constructor<? extends TestCase>  
getTestConstructor(Class<? extends TestCase> theClass)  
throws NoSuchMethodException {  
    ...  
}  
  
public static Test warning(final String message) {  
    ...  
}  
  
private static String exceptionToString(Throwable t) {  
    ...  
}  
private String fName;  
  
private Vector<Test> fTests= new Vector<Test>(10);  
  
public TestSuite() {  
}  
  
public TestSuite(final Class<? extends TestCase> theClass)  
{  
    ...  
}  
  
public TestSuite(Class<? extends TestCase> theClass, String  
name) {  
    ...  
}  
... ... ... ... ...  
}
```

Funções dependentes. Se uma função chama outra, elas devem ficar verticalmente próximas, e a que chamar deve ficar acima da que for chamada, se possível. Isso dá um fluxo natural ao programa. Se essa convenção for seguida a fim de legibilidade, os leitores poderão confiar que as declarações daquelas funções virão logo em seguida após seu uso. Considere, por exemplo, o fragmento do FitNesse na Listagem 5.5. Note como a função mais superior chama as outras abaixo dela e como elas, por sua vez, chama aquelas abaixo delas também. Isso facilita encontrar as funções chamadas e aumenta consideravelmente a legibilidade de todo o módulo.

Listagem 5-5**WikiPageResponder.java**

```
public class WikiPageResponder implements SecureResponder {
    protected WikiPage page;
    protected PageData pageData;
    protected String pageTitle;
    protected Request request;
    protected PageCrawler crawler;

    public Response makeResponse(FitNesseContext context, Request request)
        throws Exception {
        String pageName = getPageNameOrDefault(request, "FrontPage");

        loadPage(pageName, context);
        if (page == null)
            return notFoundResponse(context, request);
        else
            return makePageResponse(context);
    }

    private String getPageNameOrDefault(Request request, String defaultPageName)
    {
        String pageName = request.getResource();
        if (StringUtil.isBlank(pageName))
            pageName = defaultPageName;

        return pageName;
    }

    protected void loadPage(String resource, FitNesseContext context)
        throws Exception {
        WikiPagePath path = PathParser.parse(resource);
        crawler = context.root.getPageCrawler();
        crawler.setDeadEndStrategy(new VirtualEnabledPageCrawler());
        page = crawler.getPage(context.root, path);
        if (page != null)
            pageData = page.getData();
    }

    private Response notFoundResponse(FitNesseContext context, Request request)
        throws Exception {
        return new NotFoundResponder().makeResponse(context, request);
    }

    private SimpleResponse makePageResponse(FitNesseContext context)
        throws Exception {
        pageTitle = PathParser.render(crawler.getFullPath(page));
        String html = makeHtml(context);

        SimpleResponse response = new SimpleResponse();
        response.setMaxAge(0);
        response.setContent(html);
        return response;
    }

    ...
}
```

Além disso, esse fragmento apresenta um bom exemplo de como manter as constantes no nível apropriado [G35]. A constante “FrontPage” poderia ter sido colocada na função getPageNameOrDefault, mas isso teria ocultado uma constante bem conhecida e esperada em uma função de baixo nível. Foi melhor passar tal constante a partir do local no qual ela faz sentido para um onde ela realmente é usada.

Afinidade conceitual. Determinados bits de código querem ficar perto de outros bits. Eles possuem uma certa afinidade conceitual. Quanto maior essa afinidade, menor deve ser a distância vertical entre eles.

Como vimos, essa afinidade deve basear-se numa dependência direta, como uma função chamando outra ou uma função usando uma variável. Mas há outras causas possíveis de afinidade, que pode ser causada por um grupo de funções que efetuam uma operação parecida. Considere o pedaço de código abaixo do JUnit 4.3.1:

```
public class Assert {
    static public void assertTrue(String message, boolean condition) {
        if (!condition)
            fail(message);
    }

    static public void assertTrue(boolean condition) {
        assertTrue(null, condition);
    }

    static public void assertFalse(String message, boolean condition) {
        assertTrue(message, !condition);
    }

    static public void assertFalse(boolean condition) {
        assertFalse(null, condition);
    }
}
```



Essas funções possuem uma afinidade conceitual forte, pois compartilham de uma mesma convenção de nomes e efetuam variações de uma mesma tarefa básica. O fato de uma chamar a outra é secundário. Mesmo se não o fizessem, ainda iriam querer ficar próximas.

Ordenação vertical

De modo geral, desejamos que as chamadas das dependências da função apontem para baixo. Isto é, a função chamada deve ficar embaixo da que a chama². Isso cria um fluxo natural para baixo no módulo do código-fonte, de um nível maior para um menor.

Assim como nos artigos de jornais, esperamos que a maioria dos conceitos venha primeiro, e também que seja expressada com uma quantidade mínima de detalhes. Esperamos que os detalhes

de baixo nível venham por último. Isso nos permite passar os olhos nos arquivos-fonte e obter uma idéia de algumas das primeiras funções, sem ter de mergulhar nos detalhes. A Listagem 5.5 está organizada dessa forma. Talvez os exemplos da Listagem 15.5 (p. 263) e 3.7 (p. 50) estejam ainda melhores.

Isso é exatamente o oposto de linguagens, como Pascal, C e C++, que exigem a definição das funções, ou pelo menos a declaração, antes de serem usadas.

Formatação horizontal

Qual deve ser o tamanho de uma linha? Para responder isso, vejamos como ocorre em programas comuns.

Novamente, examinemos sete projetos diferentes. A Figura 5.2 mostra a distribuição do comprimento das linhas em todos os sete projetos. A regularidade é impressionante, cada linha fica com cerca de 45 caracteres. De fato, todo comprimento de 20 a 60 representa cerca de 1 por cento do número total de linhas. Isso são 40%! Talvez outros 30 por cento possuam menos do que 10 caracteres. Lembre-se de que é uma escala logarítmica, portanto a aparência linear do declínio gradual acima de 80 caracteres é realmente muito significante. Os programadores claramente preferem linhas curtas.

Isso sugere que devemos nos esforçar para manter nossas linhas curtas. O antigo limite de 80 de Hollerith é um pouco arbitrário, e não sou contra linhas com 100 ou mesmo 120 caracteres. Mas ultrapassar isso provavelmente é apenas falta de cuidado.

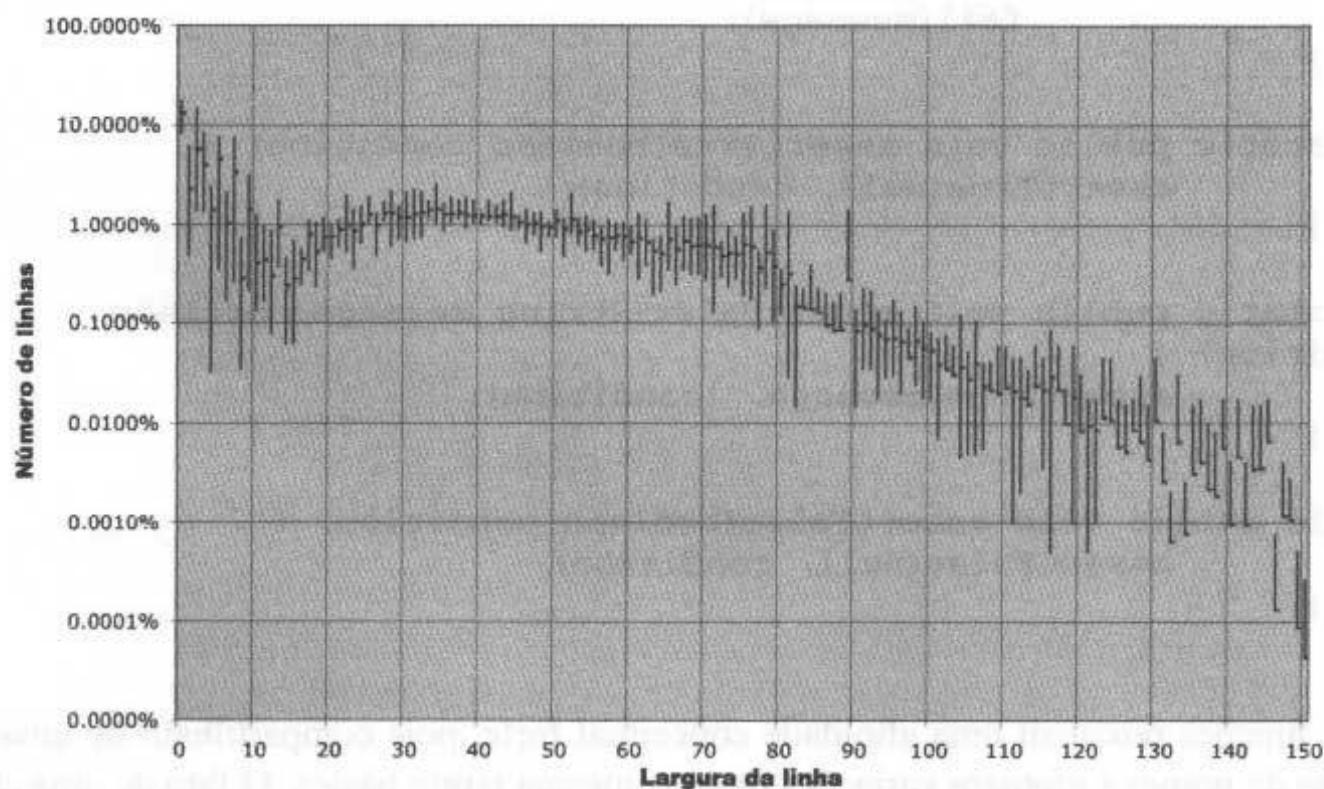


Figura 5.2:
Distribuição da largura da linha em Java

Eu costumava seguir a regra na qual jamais se deve ter de rolar a tela para a direita. Mas, hoje em dia, os monitores estão muito largos para isso, e programadores mais jovens também podem diminuir a fonte de modo que 200 caracteres caibam na tela. Não faça isso. Eu, pessoalmente, determinei 120 como meu limite.

Espaçamento e continuidade horizontal

Usamos o espaço em branco horizontal para associar coisas que estão intimamente relacionadas e para desassociar outras fracamente relacionadas. Considere a função seguinte:

```
private void measureLine(String line) {  
    lineCount++;  
    int lineSize = line.length();  
    totalChars += lineSize;  
    lineWidthHistogram.addLine(lineSize, lineCount);  
    recordWidestLine(lineSize);  
}
```

Coloquei os operadores de atribuição entre espaços em branco para destacá-los. As instruções de atribuição têm dois elementos principais e distintos: os lados esquerdo e direito. Os espaços tornam essa separação óbvia.

Por outro lado, não coloque espaços entre os nomes das funções e os parênteses de abertura. Isso porque a função e seus parâmetros estão intimamente relacionados. Separá-los iria fazer com que parecesse que não estão juntos. Eu separei os parâmetros entre parênteses na chamada da função para realçar a vírgula e mostrar que eles estão separados.

Outro uso do espaço em branco é para destacar a prioridade dos operadores.

```
public class Quadratic {  
    public static double root1(double a, double b, double c) {  
        double determinant = determinant(a, b, c);  
        return (-b + Math.sqrt(determinant)) / (2*a);  
    }  
  
    public static double root2(int a, int b, int c) {  
        double determinant = determinant(a, b, c);  
        return (-b - Math.sqrt(determinant)) / (2*a);  
    }  
  
    private static double determinant(double a, double b, double c)  
    {  
        return b*b - 4*a*c;  
    }  
}
```

Note como é fácil ler as equações. Os fatores não possuem espaços em branco entre eles porque eles têm maior prioridade. Os termos são separados por espaços em branco porque a adição e a subtração têm menor prioridade.

Infelizmente, a maioria das ferramentas para reformatação de código não faz essa distinção entre operadores e usam o mesmo espaçamento para todos. Portanto, costuma-se perder espaçamentos sutis como os acima na hora da reformatação do código.

Alinhamento horizontal

Quando eu era programador em assembly³, eu usava o alinhamento horizontal para realçar certas estruturas. Quando comecei a programar em C, C++ e, depois, em Java, continuei a tentar alinhar todos os nomes das variáveis numa série de declarações, ou todos os valores numa série de instruções de atribuição. Meu código ficava assim:

```
public class FitNesseExpediter implements ResponseSender
{
    private Socket socket;
    private InputStream input;
    private OutputStream output;
    private Request request;
    private Response response;
    private FitNesseContext context;
    protected long requestParsingTimeLimit;
    private long requestProgress;
    private long requestParsingDeadline;
    private boolean hasError;

    public FitNesseExpediter(Socket s,
                           FitNesseContext context) throws Exception
    {
        this.context = context;
        socket = s;
        input = s.getInputStream();
        output = s.getOutputStream();
        requestParsingTimeLimit = 10000;
    }
}
```

Entretanto, descobri que esse tipo de alinhamento não é prático. Ele parece enfatizar as coisas erradas e afasta meus olhos do propósito real. Por exemplo, na lista de declarações acima, você fica tentado a ler todos os nomes das variáveis sem se preocupar com seus tipos. Da mesma forma, na lista de atribuições, você se sente tentado a ler toda a lista de valores, sem se preocupar em ver o operador de atribuição. Para piorar as coisas, as ferramentas de reformatação automática geralmente eliminam esse tipo de alinhamento.

Portanto, acabei não fazendo mais esse tipo de coisa. Atualmente, prefiro declarações e atribuições não alinhadas, como mostrado abaixo, pois eles destacam uma deficiência importante. Se eu tiver listas longas que precisem ser alinhadas, *o problema está no tamanho das listas*, e não na falta de alinhamento. O comprimento da lista de declarações na FitNesseExpediter abaixo sugere que essa classe deva ser dividida.

```
public class FitNesseExpediter implements ResponseSender
{
    private Socket socket;
    private InputStream input;
    private OutputStream output;
    private Request request;
```

Quem estou tentanto enganar? Ainda sou um programador em assembly. Pode-se afastar o programador da linguagem, mas não se pode afastar a linguagem do programador!

```
private Response response;
private FitNesseContext context;
protected long requestParsingTimeLimit;
private long requestProgress;
private long requestParsingDeadline;
private boolean hasError;

public FitNesseExpediter(Socket s, FitNesseContext context) throws
Exception
{
    this.context = context;
    socket = s;
    input = s.getInputStream();
    output = s.getOutputStream();
    requestParsingTimeLimit = 10000;
}
```

Endentação

Um arquivo-fonte é mais como uma hierarquia do que algo esquematizado. Há informações pertinentes ao arquivo como um todo, às classes individuais dentro do arquivo, aos métodos dentro das classes, aos blocos dentro dos métodos e, recursivamente, aos blocos dentro de blocos. Cada nível dessa hierarquia é um escopo no qual se podem declarar nomes e no qual são interpretadas declarações e instruções executáveis.

A fim de tornar visível a hierarquia desses escopos, endentamos as linhas do código-fonte de acordo com sua posição na hierarquia. Instruções no nível do arquivo, como a maioria das declarações de classes, não são endentadas. Métodos dentro de uma classe são endentados um nível à direita dela. Implementações do método são implementadas um nível à direita da declaração do método. Implementações de blocos são implementadas um nível à direita do bloco que as contém, e assim por diante.

Os programadores dependem bastante desse esquema de endentação. Eles alinham visualmente na esquerda as linhas para ver em qual escopo elas estão. Isso lhes permite pular escopos, como de implementações de estruturas `if` e `while`, que não são relevantes no momento. Eles procuram na esquerda por novas declarações de métodos, novas variáveis e até novas classes. Sem a endentação, os programas seriam praticamente ininteligíveis para humanos.

Considere os programas seguintes sintáticamente e semanticamente idênticos:

```
public class FitNesseServer implements SocketServer { private
FitNesseContext
context; public FitNesseServer(FitNesseContext context) { this.context =
context; } public void serve(Socket s) { serve(s, 10000); } public void
serve(Socket s, long requestTimeout) { try { FitNesseExpediter sender =
new
FitNesseExpediter(s, context);
sender.setRequestParsingTimeLimit(requestTimeout); sender.start(); }
} } catch(Exception e) { e.printStackTrace(); } }
```

```

public class FitNesseServer implements SocketServer {
    private FitNesseContext context;

    public FitNesseServer(FitNesseContext context) {
        this.context = context;
    }

    public void serve(Socket s) {
        serve(s, 10000);
    }

    public void serve(Socket s, long requestTimeout) {
        try {
            FitNesseExpediter sender = new FitNesseExpediter(s,
context);
            sender.setRequestParsingTimeLimit(requestTimeout);
            sender.start();
        }
        catch (Exception e) {
            e.printStackTrace();
        }
    }
}

```

Seus olhos conseguem discernir rapidamente a estrutura do arquivo endentado. Quase instantaneamente você localiza as variáveis, os construtores, os métodos acessores (leitura e escrita, ou setter and getter) e os métodos. Bastam alguns segundos para perceber que se trata de um tipo simples de interface pra um socket, com um tempo limite. A versão sem endentação, contudo, é praticamente incompreensível sem um estudo mais profundo.

Ignorando a endentação. Às vezes, ficamos tentados a não usar a endentação em estruturas `if` curtas, loops `while` pequenos ou funções pequenas. Sempre que não resisto a essa tentação, quase sempre acabo voltando e endentando tais partes. Portanto, evito alinhar uniformemente escopos como este:

```

public class CommentWidget extends TextWidget
{
    public static final String REGEXP = "^#[^\r\n]*(?:(?:\r\n)|\n|\r)?";
}

public CommentWidget(Widget parent, String text){super(parent,
text);}
    public String render() throws Exception {return "";}
}

```

Prefiro expandir e endentar escopos, como este:

```

public class CommentWidget extends TextWidget {
    public static final String REGEXP = "^#[^\r\n]*(?:(?:\r\n)|\n|\r)?";

    public CommentWidget(Widget parent, String text) {
        super(parent, text);
    }
}

```

```
public String render() throws Exception {  
    return "";  
}  
}
```

Escopos minúsculos

De vez em quando, o corpo de uma estrutura while ou for é minúscula, como mostra abaixo. Como não gosto disso, procuro evitá-las. Quando isso não for possível, verifico se o corpo da estrutura está endentado adequadamente e entre parênteses. Inúmeras vezes já me enganei com um ponto-e-vírgula quietinho lá no final de um loop while na mesma linha. A menos que você torne esse ponto-e-vírgula visível endentando-o em sua própria linha, fica difícil visualizá-lo.

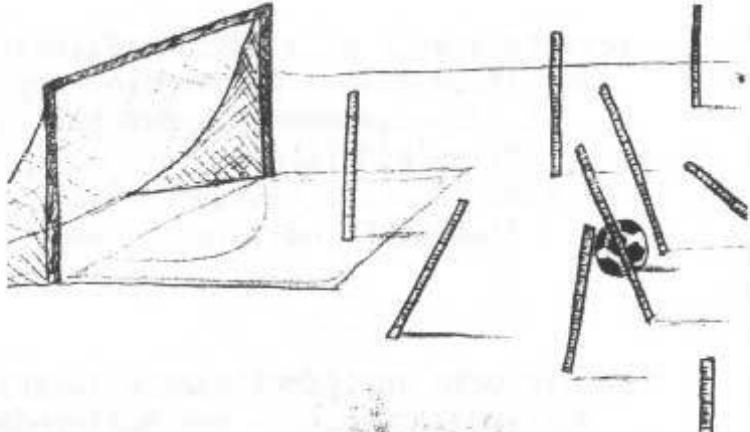
```
while (dis.read(buf, 0, readBufferSize) != -1)  
;
```

Regra de equipes

O título desse tópico faz um jogo com as palavras. Todo programador tem suas regras de formatação prediletas, mas se ele for trabalhar em equipe, as regras são dela. Uma equipe de desenvolvedores deve escolher um único estilo de formatação, e, então, todos os membros devem usá-lo. Desejamos que o software tenha um estilo consistente. Não queremos que pensem que o código foi escrito por um bando de pessoas em desacordo.

Quando entrei no projeto FitNesse em 2002, sentei com a equipe para escolher um estilo de programação. Isso levou 10 minutos. Decidimos onde colocaríamos nossas chaves, o tamanho da endentação, como nomearíamos as classes, variáveis e métodos, e assim por diante. Então, codificamos essas regras no formatador de código de nossa IDE e ficamos com ela desde então. Não eram as regras que eu preferia, mas as que foram decididas pela equipe. E como membro, tive segui-las na hora de programar no projeto FitNesse.

Lembre-se: um bom sistema de software é composto de uma série de documentos de fácil leitura. Eles precisam ter um estilo consistente e sutil. O leitor precisa poder confiar que as formatações que ele vir em um arquivo-fonte terão o mesmo significado nos outros. A última coisa que queremos é adicionar mais complexidade ao código-fonte programando-o com um monte de estilos diferentes.



Regras de formatação do Uncle Bob

As regras que uso são muito simples e estão ilustradas no código da Listagem 5.6.

Considere isso um exemplo de como o código é o melhor documento padrão em programação.

Listagem 5-6**CodeAnalyzer.java**

```
public class CodeAnalyzer implements JavaFileAnalysis {
    private int lineCount;
    private int maxLineWidth;
    private int widestLineNumber;
    private LineWidthHistogram lineWidthHistogram;
    private int totalChars;

    public CodeAnalyzer() {
        lineWidthHistogram = new LineWidthHistogram();
    }

    public static List<File> findJavaFiles(File parentDirectory) {
        List<File> files = new ArrayList<File>();
        findJavaFiles(parentDirectory, files);
        return files;
    }

    private static void findJavaFiles(File parentDirectory, List<File> files) {
        for (File file : parentDirectory.listFiles()) {
            if (file.getName().endsWith(".java"))
                files.add(file);
            else if (file.isDirectory())
                findJavaFiles(file, files);
        }
    }

    public void analyzeFile(File javaFile) throws Exception {
        BufferedReader br = new BufferedReader(new FileReader(javaFile));
        String line;
        while ((line = br.readLine()) != null)
            measureLine(line);
    }

    private void measureLine(String line) {
        lineCount++;
        int lineSize = line.length();
        totalChars += lineSize;
        lineWidthHistogram.addLine(lineSize, lineCount);
        recordWidestLine(lineSize);
    }
}
```

Listagem 5-6 (continuação)**CodeAnalyzer.java**

```
private void recordWidestLine(int lineSize) {
    if (lineSize > maxLineWidth) {
        maxLineWidth = lineSize;
        widestLineNumber = lineCount;
    }
}

public int getLineCount() {
    return lineCount;
}

public int getMaxLineWidth() {
    return maxLineWidth;
}

public int getWidestLineNumber() {
    return widestLineNumber;
}

public LineWidthHistogram getLineWidthHistogram() {
    return lineWidthHistogram;
}

public double getMeanLineWidth() {
    return (double)totalChars/lineCount;
}

public int getMedianLineWidth() {
    Integer[] sortedWidths = getSortedWidths();
    int cumulativeLineCount = 0;
    for (int width : sortedWidths) {
        cumulativeLineCount += lineCountForWidth(width);
        if (cumulativeLineCount > lineCount/2)
            return width;
    }
    throw new Error("Cannot get here");
}

private int lineCountForWidth(int width) {
    return lineWidthHistogram.getLinesforWidth(width).size();
}

private Integer[] getSortedWidths() {
    Set<Integer> widths = lineWidthHistogram.getWidths();
    Integer[] sortedWidths = (widths.toArray(new Integer[0]));
    Arrays.sort(sortedWidths);
    return sortedWidths;
}
```

Objetos e Estruturas de Dados



Há um motivo para declararmos nossas variáveis como privadas. Não queremos que ninguém dependa delas. Desejamos ter a liberdade para alterar o tipo ou a implementação, seja por capricho ou impulso. Por que, então, tantos programadores adicionam automaticamente métodos de acesso (escrita, ou *setters*, e leitura, ou *getters*) em seus objetos, expondo suas variáveis privadas como se fossem públicas?

Abstração de dados

Considere a diferença entre as listagens 6.1 e 6.2. Ambas representam os dados de um ponto no plano cartesiano. Um expõe sua implementação e o outro a esconde completamente.

Listagem 6-1**Caso concreto**

```
public class Point {
    public double x;
    public double y;
}
```

Listagem 6-2**Caso abstrato**

```
public interface Point {
    double getX();
    double getY();
    void setCartesian(double x, double y);
    double getR();
    double getTheta();
    void setPolar(double r, double theta);
}
```

O belo da Listagem 6.2 é que não há como dizer se a implementação possui coordenadas retangulares ou polares. Pode não ser nenhuma! E ainda assim a interface representa de modo claro uma estrutura de dados.

Mas ela faz mais do que isso. Os métodos exigem uma regra de acesso. Você pode ler as coordenadas individuais independentemente, mas deve configurá-las juntas como uma operação atômica.

A Listagem 6.1, por outro lado, claramente está implementada em coordenadas retangulares, e nos obriga a manipulá-las independentemente. Isso expõe a implementação. De fato, ela seria exposta mesmo se as variáveis fossem privadas e estivéssemos usando métodos únicos de escrita e leitura de variáveis.

Ocultar a implementação não é só uma questão de colocar uma camada de funções entre as variáveis. É uma questão de ! Uma classe não passa suas variáveis simplesmente por meio de métodos de escrita e leitura. Em vez disso, ela expõe interfaces abstratas que permite aos usuários manipular a *essência* dos dados, sem precisar conhecer a implementação.

Considere as listagens 6.3 e 6.4. A primeira usa termos concretos para comunicar o nível de combustível de um veículo, enquanto a segunda faz o mesmo, só que usando . No caso concreto, você tem certeza de que ali estão apenas métodos acessores (escrita e leitura, ou *getter* e *setter*) de variáveis. No caso abstrato, não há como saber o tipo dos dados.

Listagem 6-3**Veículo concreto**

```
public interface Vehicle {
    double getFuelTankCapacityInGallons();
    double getGallonsOfGasoline();
}
```

Listagem 6-4**Veículo abstrato**

```
public interface Vehicle {  
    double getPercentFuelRemaining();  
}
```

Em ambos os casos acima, o segundo é preferível. Não queremos expor os detalhes de nossos dados. Queremos expressar nossos dados de forma abstrata. Isso não se consegue meramente através de interfaces e/ou métodos de escrita e leitura. É preciso pensar bastante na melhor maneira de representar os dados que um objeto contenha. A pior opção é adicionar levianamente métodos de escrita e leitura.

Anti-simetria data/objeto

Esses dois exemplos mostram a diferença entre objetos e estruturas de dados. Os objetos usam abstrações para esconder seus dados, e expõem as funções que operam em tais dados. As estruturas de dados expõem seus dados e não possuem funções significativas. Leia este parágrafo novamente.

Note a natureza complementar das duas definições. Elas são praticamente opostas. Essa diferença pode parecer trivial, mas possui grandes implicações.

Considere, por exemplo, a classe `shape` procedural na Listagem 6.5. A classe `Geometry` opera em três classes `shape` que são simples estruturas de dados sem qualquer atividade. Todas as ações estão na classe `Geometry`.

Listagem 6-5**Classe shape procedural**

```
public class Square {  
    public Point topLeft;  
    public double side;  
}  
  
public class Rectangle {  
    public Point topLeft;  
    public double height;  
    public double width;  
}  
  
public class Circle {  
    public Point center;  
    public double radius;  
}  
  
public class Geometry {  
    public final double PI = 3.141592653589793;  
  
    public double area(Object shape) throws NoSuch:  
    {  
        if (shape instanceof Square) {  
            Square s = (Square)shape;  
            return s.side * s.side;  
        }  
    }  
}
```

Listagem 6-5 (continuação)

Classe shape procedimental

```
        else if (shape instanceof Rectangle) {
            Rectangle r = (Rectangle)shape;
            return r.height * r.width;
        }
        else if (shape instanceof Circle) {
            Circle c = (Circle)shape;
            return PI * c.radius * c.radius;
        }
        throw new NoSuchShapeException();
    }
```

Programadores de orientação a objeto talvez torçam o nariz e reclamem que isso é procedural—e estão certos. Mas nem sempre. Imagine o que aconteceria se adicionássemos uma função `perimeter()` à `Geometry`. As classes `shape` não seriam afetadas! Assim como quaisquer outras classes que dependessem delas!

Por outro lado, se adicionarmos uma nova classe shape, teremos de alterar todas as funções em Geometry. Leia essa frase novamente. Note que as duas situações são completamente opostas.

Agora, considere uma solução orientada a objeto na Listagem 6.6. O método `área()` é *polifórmico*. Não é necessária a classe `Geometry`. Portanto, se eu adicionar uma nova forma, nenhuma das funções existentes serão afetadas, mas se eu adicionar uma nova função, todas as classes `shape` deverão ser alteradas¹.

Listagem 6-6

Classes shape polifórmicas

```
public class Square implements Shape {  
    private Point topLeft;  
    private double side;  
  
    public double area() {  
        return side*side;  
    }  
  
}  
  
public class Rectangle implements Shape {  
    private Point topLeft;  
    private double height;  
    private double width;  
  
    public double area() {  
        return height * width;  
    }  
}
```

¹ Desenvolvedores orientados a objeto experiente conhecem outras maneiras de se contornar isso. O padrão Visitor é dual-dispatch, por exemplo.

Listagem 6-6 (continuação)**Classes shape polifôrmicas**

```
public class Circle implements Shape {
    private Point center;
    private double radius;
    public final double PI = 3.141592653589793;

    public double area() {
        return PI * radius * radius;
    }
}
```

Novamente, vemos que a natureza complementar dessas duas definições: elas são praticamente opostas! Isso expõe a dicotomia fundamental entre objetos e estruturas de dados:

O código procedural (usado em estruturas de dados) facilita a adição de novas funções sem precisar alterar as estruturas de dados existentes. O código orientado a objeto (OO), por outro lado, facilita a adição de novas classes sem precisar alterar as funções existentes.

O inverso também é verdade:

O código procedural dificulta a adição de novas estruturas de dados, pois todas as funções teriam de ser alteradas. O código OO dificulta a adição de novas funções, pois todas as classes teriam de ser alteradas.

Portanto, o que é difícil para a OO é fácil para o procedural, e o que é difícil para o procedural é fácil para a OO!

Em qualquer sistema complexo haverá vezes nas quais desejaremos adicionar novos tipos de dados em vez de novas funções. Para esses casos, objetos e OO são mais apropriados. Por outro lado, também haverá vezes nas quais desejaremos adicionar novas funções em vez de tipos de dados. Neste caso, estruturas de dados e código procedural são mais adequados.

Programadores experientes sabem que a ideia de que tudo é um objeto é *um mito*. Às vezes, você realmente *deseja* estruturas de dados simples com procedimentos operando nelas.

A lei de Demeter

Há uma nova heurística muito conhecida chamada Lei de Demeter²: um módulo não deve enxergar o interior dos objetos que ele manipula. Como vimos na seção anterior, os objetos escondem seus dados e expõem as operações. Isso significa que um objeto não deve expor sua estrutura interna por meio dos métodos acessores, pois isso seria expor, e não ocultar, sua estrutura interna.

Mais precisamente, a Lei de Demeter diz que um método *f* de uma classe *C* só deve chamar os métodos de:

- *C*
- Um objeto criado por *f*
- Um objeto passado como parâmetro para *f*
- Um objeto dentro de uma instância da variável *C*

² http://en.wikipedia.org/wiki/Law_of_Demeter

O método *não* deve chamar os métodos em objetos retornados por qualquer outra das funções permitidas. Em outras palavras, fale apenas com conhecidos, não com estranhos.

O código³ seguinte parece violar a Lei de Demeter (entre outras coisas), pois ele chama a função `getScratchDir()` no valor retornado de `getOptions()` e, então, chama `getAbsolutePath()` no valor retornado de `getScratchDir()`.

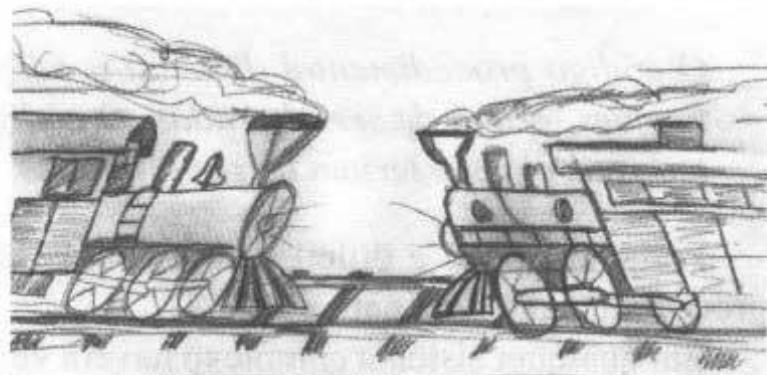
```
final String outputDir = ctxt.getOptions().getScratchDir().  
getAbsolutePath();
```

Carrinhos de trem

Esse tipo de código costuma ser chamador de *carrinho de trem*, pois parece com um monte de carrinhos de trem acoplados. Cadeias de chamadas como essa geralmente são consideradas descuidadas e devem ser evitadas [G36]. Na maioria das vezes é melhor dividi-las assim:

```
Options opts = ctxt.getOptions();  
File scratchDir = opts.getScratchDir();  
final String outputDir = scratchDir.getAbsolutePath();
```

Esses dois pedaços de código violam a Lei de Demeter? Certamente módulo que os contém sabe que o objeto `ctxt` possui opções (*options*), que contêm um diretório de rascunho (*scratchDir*), que tem um caminho absoluto (*AbsolutePath*). É muito conhecimento para uma função saber. A função de chamada sabe como navegar por muitos objetos diferentes.



Se isso é uma violação da Lei de Demeter depende se `ctxt`, `Options` e `ScracthDir` são ou não objetos ou estruturas de dados. Se forem objetos, então sua estrutura interna deveria estar oculta ao invés de exposta, portanto o conhecimento de seu interior é uma violação clara da lei. Por outro lado, se forem apenas estruturas de dados sem atividades, então eles naturalmente expõem suas estruturas internas, portanto aqui não se aplica a lei.

O uso de funções de acesso confunde essas questões. Se o código tiver sido escrito como abaixo, então provavelmente não estariamos perguntando sobre cumprimento ou não da lei.

```
final String outputDir = ctxt.options.scratchDir.getAbsolutePath();
```

Essa questão seria bem menos confusa se as estruturas de dados simplesmente tivessem variáveis públicas e nenhuma função, enquanto os objetos tivessem apenas variáveis privadas e funções públicas. Entretanto, há frameworks e padrões (*e.g.*, “beans”) que exigem que mesmo estruturas de dados simples tenham métodos acessores e de alteração.

³. Está em algum lugar no framework do Apache.

Híbridos

De vez em quando, essa confusão leva a estruturas híbridas ruins que são metade objeto e metade estrutura de dados. Elas possuem funções que fazem algo significativo, e também variáveis ou métodos de acesso e de alteração públicos que, para todos os efeitos, tornam públicas as variáveis privadas, incitando outras funções externas a usarem tais variáveis da forma como um programa procedural usaria uma estrutura de dados⁴.

Esses híbridos dificultam tanto a adição de novas funções como de novas estruturas de dados. Eles são a pior coisa em ambas as condições. Evite criá-los. Eles indicam um modelo confuso cujos autores não tinham certeza – ou pior, não sabiam – se precisavam se proteger de funções ou tipos.

Estruturas ocultas

E se ctxt, options e scratchDir forem objetos com ações reais? Então, como os objetos devem ocultar suas estruturas internas, não deveríamos ser capazes de navegar por eles. Então, como conseguiríamos o caminho absoluto de scratchDir ('diretório de rascunho')?

```
ctxt.getAbsolutePathOfScratchDirectoryOption();
```

ou

```
ctx.getScratchDirectoryOption().getAbsolutePath()
```

A primeira opção poderia levar a uma abundância de métodos no objeto ctxt. A segunda presume que `getScratchDirectoryOption()` retorna uma estrutura de dados, e não um objeto. Nenhuma das opções parece boa.

Se ctxt for um objeto, devemos dizê-lo para fazer algo; não devemos perguntá-lo sobre sua estrutura interna. Por que queremos o caminho absoluto de scratchDir? O que faremos com ele? Considere o código, muitas linhas abaixo, do mesmo módulo:

```
String outFile = outputDir + "/" + className.replace('.', '/') +
    ".class";
FileOutputStream fout = new FileOutputStream(outFile);
BufferedOutputStream bos = new BufferedOutputStream(fout);
```

A mistura adicionada de diferentes níveis de detalhes [G34][G6] é um pouco confusa. Pontos, barras, extensão de arquivos e objetos `File` não devem ser misturados entre si e nem com o código que os circunda. Ignorando isso, entretanto, vimos que a intenção de obter o caminho absoluto do diretório de rascunho era para criar um arquivo de rascunho de um determinado nome. Então, e se disséssemos ao objeto ctxt para fazer isso?

```
BufferedOutputStream bos = ctxt.createScratchFileStream(classFileName);
```

That seems like a reasonable thing for an object to do! Isso permite ao ctxt esconder sua estrutura interna e evitar que a função atual viole a Lei de Demeter ao navegar por objetos os quais ela não deveria enxergar.

4. Às vezes chama-se de *Feature Envy* em [Refatoração].

Objetos de transferência de dados

A forma perfeita de uma estrutura de dados é uma classe com variáveis públicas e nenhuma função.

Às vezes, chama-se isso de objeto de transferência de dados, ou DTO (sigla em inglês). Os DTOs são estruturas muitos úteis, especialmente para se comunicar com bancos de dados ou analisar sintaticamente mensagens provenientes de sockets e assim por diante. Eles costumam se tornar os primeiros numa série de estágios de tradução que convertem dados brutos num banco de dados em objetos no código do aplicativo.

De alguma forma mais comum é o formulário “bean” exibido na Listagem 6.7. Os beans têm variáveis privadas manipuladas por métodos de escrita e leitura. O aparente encapsulamento dos beans parece fazer alguns puristas da OO sentirem-se melhores, mas geralmente não oferece vantagem alguma.

Listagem 6-7

address.java

```
public class Address {  
    private String street;  
    private String streetExtra;  
    private String city;  
    private String state;  
    private String zip;  
  
    public Address(String street, String streetExtra,  
                  String city, String state, String zip) {  
        this.street = street;  
        this.streetExtra = streetExtra;  
        this.city = city;  
        this.state = state;  
        this.zip = zip;  
    }  
  
    public String getStreet() {  
        return street;  
    }  
  
    public String getStreetExtra() {  
        return streetExtra;  
    }  
  
    public String getCity() {  
        return city;  
    }  
  
    public String getState() {  
        return state;  
    }  
  
    public String getZip() {  
        return zip;  
    }  
}
```

O Active Record

Os Active Records são formas especiais de DTOs. Eles são estruturas de dados com variáveis públicas (ou acessadas por Beans); mas eles tipicamente possuem métodos de navegação, como `save` (salvar) e `find` (buscar). Esses Active Records são traduções diretas das tabelas de bancos de dados ou de outras fontes de dados.

Infelizmente, costumamos encontrar desenvolvedores tentando tratar essas estruturas de dados como se fossem objetos, colocando métodos de regras de negócios neles. Isso é complicado, pois cria um híbrido entre uma estrutura de dados e um objeto.

A solução, é claro, é tratar o Record Active como uma estrutura de dados e criar objetos separados que contenham as regras de negócio e que ocultem seus dados internos (que provavelmente são apenas instâncias do Active Record).

Conclusão

Os objetos expõem as ações e ocultam os dados. Isso facilita a adição de novos tipos de objetos sem precisar modificar as ações existentes e dificulta a inclusão de novas atividades em objetos existentes. As estruturas de dados expõem os dados e não possuem ações significativas. Isso facilita a adição de novas ações às estruturas de dados existentes e dificulta a inclusão de novas estruturas de dados em funções existentes.

Em um dado sistema, às vezes, desejaremos flexibilidade para adicionar novos tipos de dados, e, portanto, optaremos por objetos. Em outras ocasiões, desejaremos querer flexibilidade para adicionar novas ações, e, portanto, optaremos tipos de dados e procedimentos.

Bons desenvolvedores de software entendem essas questões sem preconceito e selecionam a abordagem que melhor se aplica no momento.

Bibliografia

[Refatoração] *Refatoração - Aperfeiçoando o Projeto de Código Existente*, Martin Fowler et al., Addison-Wesley, 1999.

Tratamento de Erro

por Michael Feathers



Pode parecer estranho ter uma seção sobre tratamento de erro num livro sobre código limpo, mas essa tarefa é uma das quais todos temos de fazer quando programamos. A entrada pode estar errada e os dispositivos podem falhar. Em suma, as coisas podem dar errado, e quando isso ocorre, nós, como programadores, somos responsáveis por certificar que nosso código faça o que seja preciso fazer.

A conexão com um código limpo, entretanto, deve ser clara. O tratamento de erro domina completamente muitos códigos-fonte. Quando digo “domina”, não quero dizer que eles só fazem tratamento de erro, mas que é quase impossível ver o que o código faz devido a tantos tratamentos de erros espalhados. Esse recurso é importante, *mas se obscurecer a lógica, está errado*.

Neste capítulo ressaltarei uma série de técnicas e considerações que você pode usar para criar um código que seja limpo e robusto, que trate de erros com elegância e estilo.

Use exceções em vez de retornar códigos

Num passado longínquo havia muitas linguagens que não suportavam exceções. Nelas, as técnicas para tratar e informar erros era limitada. Ou você criava uma flag de erro ou retornava um código de erro que o chamador pudesse verificar. O código na Listagem 7.1 ilustra essas abordagens.

Listagem 7-1

DeviceController.java

```
public class DeviceController {
    ...
    public void sendShutDown() {
        DeviceHandle handle = getHandle(DEV1);
        // Check the state of the device
        if (handle != DeviceHandle.INVALID) {
            // Save the device status to the record field
            retrieveDeviceRecord(handle);
            // If not suspended, shut down
            if (record.getStatus() != DEVICE_SUSPENDED) {
                pauseDevice(handle);
                clearDeviceWorkQueue(handle);
                closeDevice(handle);
            } else {
                logger.log("Device suspended. Unable to shut down");
            }
        } else {
            logger.log("Invalid handle for: " + DEV1.toString());
        }
    }
    ...
}
```

O problema era que essas técnicas entupiam o chamador, que devia verificar erros imediatamente após a chamada. Infelizmente, facilmente se esqueciam de fazer isso. Por esse motivo, é melhor lançar uma exceção quando um erro for encontrado. O código de chamada fica mais limpo e sua lógica não fica ofuscada pelo tratamento de erro.

A Listagem 7.2 mostra o código depois de termos optado por lançar exceções em métodos que podem detectar erros.

Listagem 7-2

DeviceController.java (com exceções)

```
public class DeviceController {
    ...
    public void sendShutDown() {
        try {
            tryToShutDown();
        } catch (DeviceShutDownError e) {
            logger.log(e);
        }
    }
}
```

Listagem 7-2 (continuação):**DeviceController.java (com exceções)**

```

private void tryToShutdown() throws DeviceShutDownError {
    DeviceHandle handle = getHandle(DEV1);
    DeviceRecord record = retrieveDeviceRecord(handle);

    pauseDevice(handle);
    clearDeviceWorkQueue(handle);
    closeDevice(handle);
}

private DeviceHandle getHandle(DeviceID id) {
    ...
    throw new DeviceShutDownError("Invalid handle for: " + id.toString());
    ...
}

...
}

```

Observe como fica muito mais claro. Isso não é apenas uma questão estética. O código fica melhor porque duas preocupações que estavam intrincadas, o algoritmo para o desligamento do dispositivo e o tratamento de erro, agora estão separadas. Você pode pegar cada uma delas e estudá-las independentemente.

Crie primeiro sua estrutura try-catch-finally

Uma das coisas mais interessantes sobre exceções é que elas definem um escopo dentro de seu programa. Ao executar o código na parte try da estrutura try...catch...finally, você declara que aquela execução pode ser cancelada a qualquer momento e, então, continuar no catch.

De certa forma, os blocos try são como transações. Seu catch tem de deixar seu programa num estado consistente, não importa o que aconteça no try. Por essa razão, uma boa prática é começar com uma estrutura try...catch...finally quando for escrever um código que talvez lance exceções. Isso lhe ajuda a definir o que o usuário do código deve esperar, independente do que ocorra de errado no código que é executado no try.

Vejamos um exemplo. Precisamos criar um código que acesse um arquivo e consulte alguns objetos em série.

Começamos com um teste de unidade que mostra como capturar uma exceção se o arquivo não existir:

```

@Test(expected = StorageException.class)
public void retrieveSectionShouldThrowOnInvalidFileName() {
    sectionStore.retrieveSection("invalid - file");
}

```

O teste nos leva a cria esse stub:

```

public List<RecordedGrip> retrieveSection(String sectionName) {
    // retorno ficticio ate que tenhamos uma implementacao real
    return new ArrayList<RecordedGrip>();
}

```

Nosso teste falha porque ele não lança uma exceção. Em seguida, mudamos nossa implementação de modo a tentar acessar um arquivo inválido. Essa operação lança uma exceção:

```
public List<RecordedGrip> retrieveSection(String sectionName) {
    try {
        FileInputStream stream = new FileInputStream(sectionName)
    } catch (Exception e) {
        throw new StorageException("retrieval error", e);
    }
    return new ArrayList<RecordedGrip>();
}
```

Nosso teste funciona agora porque capturamos a exceção. Neste momento, podemos refatorar. Podemos reduzir o tipo de execução que capturamos para combinar com aquele que realmente é lançado pelo construtor `FileInputStream: FileNotFoundException`:

```
public List<RecordedGrip> retrieveSection(String sectionName) {
    try {
        FileInputStream stream = new FileInputStream(sectionName);
        stream.close();
    } catch (FileNotFoundException e) {
        throw new StorageException("retrieval error", e);
    }
    return new ArrayList<RecordedGrip>();
}
```

Agora que definimos o escopo com uma estrutura `try...catch`, podemos usar o TDD para construir o resto da lógica que precisamos, que será adicionada na criação do `FileInputStream` do `close` e poderá fingir que nada de errado aconteceu.

Experimente criar testes que forcão exceções e, então, adicione a ação ao seu tratador para cumprir seus testes. Isso fará com que você crie primeiro o escopo de transação do bloco `try` e lhe ajudará a manter essa natureza de transação daquele escopo.

Use exceções não verificadas

A discussão acabou. Por anos, programadores Java têm discutido sobre as vantagens e desvantagens de exceções verificadas. Quando a verificação exceções surgiu com a primeira versão do Java, parecia uma ótima ideia. A assinatura de todo método listaria todas as exceções que ele passaria ao seu chamador. Ademais, essas exceções eram parte do tipo do método. Seu código literalmente não seria compilado se a assinatura não fosse a mesma da que seu código podia fazer.

Naquela época, pensamos que exceções verificadas fosse uma ideia ótima; e era, elas tinham *algumas* vantagens. Entretanto, ficou claro agora que elas não são necessárias para a produção de um software robusto. O C# não verifica exceções, e, apesar das tentativas, nem o C++. Nem mesmo Python ou Ruby. Ainda assim é possível criar um software robusto em todas essas linguagens, porque, nesse caso, temos de decidir se realmente as exceções verificadas valem o preço que se paga.

Que preço? O de verificar exceções é a violação do Princípio de Aberto-Fechado¹.

Se você lançar uma exceção a ser verificada a partir de um método em seu código e o

catch estiver três níveis acima, *será preciso declará-la na assinatura de cada método entre você e o catch*. Isso significa que uma modificação em um nível mais baixo do software pode forçar a alteração de assinaturas em muitos níveis mais altos. Os módulos alterados podem ser reconstruídos e redistribuídos, mesmo que nada inerente a eles tenha sido mudado.

Considere a hierarquia de chamadas de um sistema grande. As funções no topo chamam as abaixo delas, que chamam outras abaixo delas e *ad infinitum*. Agora digamos que uma das funções dos níveis mais baixos seja modificada de uma forma que ela deva lançar uma exceção. Se essa exceção for verificada, então a assinatura da função deverá adicionar uma instrução throws. Mas isso significa que cada função que chamar nossa função modificada também deverá ser alterada para capturar a nova exceção ou anexar a instrução throws apropriada a sua assinatura. *Ad infinitum*. O resultado aninhado é uma cascata de alterações que vão desde os níveis mais baixo do software até o mais alto! Quebra-se o encapsulamento, pois todas as funções no caminho de um lançamento (*throw*) devem enxergar os detalhes daquela exceção de nível mais baixo. Segundo o propósito de exceções de que elas lhe permitem tratar erros distantes, é uma pena que as exceções verificadas quebrem dessa forma o encapsulamento.

As exceções verificadas podem às vezes ser úteis se você estiver criando uma biblioteca crítica: é preciso capturá-las. Mas no desenvolvimento geral de aplicativo, os custos da dependência superam as vantagens.

Forneça exceções com contexto

Cada exceção lançada deve fornecer contexto o suficiente para determinar a fonte e a localização de um erro. Em Java, você pode pegar um *stack trace* de qualquer exceção; entretanto, ele não consegue lhe dizer o objetivo da operação que falhou.

Crie mensagens de erro informativas e as passe juntamente com as exceções. Mencione a operação que falhou e o tipo da falha. Se estiver registrando as ações de seu aplicativo, passe informações suficientes para registrar o erro de seu catch.

Defina as classes de exceções segundo as necessidades do chamador

Há muitas formas de classificar erros. Pode ser pela origem: eles vieram desse componente ou daquele? Pelo tipo: são falhas de dispositivos, de redes ou erros de programação? Entretanto, quando definimos as classes de exceção num aplicativo, nossa maior preocupação deveria ser *como elas são capturadas*.

Vejamos um exemplo de uma classificação ruim de exceção. Aqui, há uma estrutura try...catch...finally para uma chamada a uma biblioteca de outro fabricante. Ela cobre todas as exceções que a chamada talvez lance:

```
ACMEPort port = new ACMEPort(12);

try {
    port.open();
} catch (DeviceResponseException e) {
    reportPortError(e);
    logger.log("Device response exception", e);
} catch (ATM1212UnlockedException e) {
    reportPortError(e);
```

```
        logger.log("Unlock exception", e);
    } catch (GMXError e) {
        reportPortError(e);
        logger.log("Device response exception");
    } finally {
        ...
    }
```

A estrutura possui muita duplicação, e não deveríamos estar surpresos. Na maioria dos casos de tratamento de exceções, o que fazemos é relativamente padrão, independente da situação no momento. Temos de registrar um erro e nos certificar que podemos prosseguir.

Neste caso, como sabemos que a tarefa que estamos fazendo é basicamente a mesma independente da exceção, podemos simplificar nosso código consideravelmente. Para isso, pegamos a API que estamos chamando e garantimos que ela retorne um tipo comum de exceção.

```
LocalPort port = new LocalPort(12);
try {
    port.open();
} catch (PortDeviceFailure e) {
    reportError(e);
    logger.log(e.getMessage(), e);
} finally {
    ...
}
```

Nossa classe `LocalPort` é um simples wrapper (“empacotador”) que captura e traduz as exceções lançadas pela classe `ACMEPort`:

```
public class LocalPort {
    private ACMEPort innerPort;

    public LocalPort(int portNumber) {
        innerPort = new ACMEPort(portNumber);
    }

    public void open() {
        try {
            innerPort.open();
        } catch (DeviceResponseException e) {
            throw new PortDeviceFailure(e);
        } catch (ATM1212UnlockedException e) {
            throw new PortDeviceFailure(e);
        } catch (GMXError e) {
            throw new PortDeviceFailure(e);
        }
    }
}
```

Wrappers como o que definimos para a `ACMEPort` podem ser muito úteis. Na verdade, empacotar APIs de terceiros é a melhor prática que existe. Ao fazer isso, você minimiza as dependências nelas: você pode escolher migrar para uma biblioteca diferente no futuro sem

muitos problemas. Empacotar também facilita a simulação de chamadas de terceiros quando for testar seu próprio código.

Uma última vantagem de empacotar (wrapping) é que você não fica preso às escolhas do modelo de API de um fornecedor em particular. Você pode definir a API que preferir. No exemplo anterior, definimos um único tipo de exceção para a falha do dispositivo `port` e descobrimos que poderíamos escrever um código muito mais limpo.

Geralmente, uma única classe de exceção está bom para uma parte específica do código. As informações enviadas com a exceção podem distinguir os erros. Use classes diferentes apenas se houver casos em que você queira capturar uma exceção e permitir que a outra passe normalmente.

Defina o fluxo normal

Se você seguir os conselhos das seções anteriores, acabará com uma boa quantidade de divisão entre sua lógica do negócio e seu tratamento de erro. A maioria de seu código começará a parecer um algoritmo limpo e sem apetrechos. Entretanto, esse processo eleva ao máximo a detecção de erro em seu programa. Você empacota suas APIs de modo que você possa lançar suas próprias exceções e definir um controlador acima de seu código para que você possa lidar com qualquer computação cancelada. Na maioria das vezes, essa é uma abordagem ótima, mas há situações nas quais você talvez não queira cancelar.



Vejamos um exemplo. Abaixo está um código confuso que soma as despesas em um aplicativo de finanças:

```
try {
    MealExpenses expenses = expenseReportDAO.getMeals(employee.
getID());
    m_total += expenses.getTotal();
} catch(MealExpensesNotFound e) {
    m_total += getMealPerDiem();
}
```

Neste negócio, se as refeições (meals) forem um custo, elas se tornam parte do total. Caso contrário, o funcionário (employee) recebe uma quantia para ajuda de custos (PerDiem) pela refeição daquele dia. A exceção confunde a lógica.

Não seria melhor se não tivéssemos de lidar com o caso especial? Dessa forma, nosso código seria muito mais simples. Ele ficaria assim:

```
MealExpenses expenses = expenseReportDAO.getMeals(employee.getID());
m_total += expenses.getTotal();
```

Podemos tornar o código mais simples? Parece que sim. Podemos alterar a `ExpenseReportDAO` de modo que ela sempre retorne um objeto `MealExpense`. Se não houver gastos com refeições,

ela retorna um objeto `MealExpense` que retorna a ajuda de custos como seu total:

```
public class PerDiemMealExpenses implements MealExpenses {
    public int getTotal() {
        // retorna a ajuda de custos padrao
    }
}
```

Isso se chama o Padrão Special Case ('padrão do caso especial'), de Fowler. Você cria uma classe ou configura um objeto de modo que ele trate de um caso especial para você. Ao fazer isso, o código do cliente não precisa lidar com o comportamento diferente. Este fica encapsulado num objeto de caso especial.

Não retorne null

Acho que qualquer discussão sobre tratamento de erro deveria incluir as coisas que fazemos que levam a erros. A primeira da lista seria retornar `null`. Já perdi a conta dos aplicativos que já vi que em quase toda linha verificava por `null`. Abaixo está um exemplo:

```
public void registerItem(Item item) {
    if (item != null) {
        ItemRegistry registry = persistentStore.getItemRegistry();
        if (registry != null) {
            Item existing = registry.getItem(item.getID());
            if (existing.getBillingPeriod().hasRetailOwner()) {
                existing.register(item);
            }
        }
    }
}
```

Se você trabalhar num código-fonte como esse, ele pode não parecer tão ruim assim para você, mas ele é! Quando retornamos `null`, basicamente estamos criando mais trabalho para nós mesmos e jogando problemas em cima de nossos chamadores. Só basta esquecer uma verificação `null` para que o aplicativo fique fora de controle.

Percebeu que não havia uma verificação de `null` na segunda linha do `if` aninhado? O que teria acontecido em tempo de execução se `persistentStore` fosse `null`? Teríamos uma `NullPointerException` em tempo de execução, e ou alguém está capturando-a no nível mais alto ou não. Em ambos os casos isso é péssimo. O que você faria exatamente em resposta a um lançamento de `NullPointerException` das profundezas de seu aplicativo?

É fácil dizer que o problema com o código acima é a falta de uma verificação de `null`, mas, na verdade, o problema é que ele tem muitos. Se você ficar tentado a retornar `null` de um método, em vez disso, considere lançar uma exceção ou retornar um objeto SPECIAL CASE. Se estiver chamando um método que retorne `null` a partir de uma API de terceiros, considere empacotá-lo com um método que lance uma exceção ou retorne um objeto de caso especial.

Em muitos casos, objetos de casos especiais são uma solução fácil. Imagine que seu código fosse assim:

```
List<Employee> employees = getEmployees();
if (employees != null) {
    for (Employee e : employees) {
```

```
        totalPay += e.getPay();
    }
}
```

Neste momento, `getEmployees` pode retornar `null`, mas ele precisa? Se alterássemos `getEmployee` de modo que ele retornasse uma lista vazia, poderíamos limpar o código:

```
List<Employee> employees = getEmployees();
for(Employee e : employees) {
    totalPay += e.getPay();
}
```

Felizmente, Java possui o `Collections.emptyList()`, e ele retorna uma lista predefinida e imutável que podemos usar para esse propósito:

```
public List<Employee> getEmployees() {
if( ... there are no employees ... )
    return Collections.emptyList();
}
```

Se programar dessa forma, você minimizará a chance de `NullPointerExceptions` e seu código será mais limpo.

Não passe null

Retornar `null` dos métodos é ruim, mas passar `null` para eles é pior. A menos que esteja trabalhando com uma API que espere receber `null`, você deve evitar passá-lo em seu código sempre que possível.

Vejamos um exemplo do porquê. Abaixo está um método simples que calcula a distância entre dois pontos:

```
public class MetricsCalculator
{
    public double xProjection(Point p1, Point p2) {
        return (p2.x - p1.x) * 1.5;
    }
    ...
}
```

O que acontece quando alguém passa `null` como parâmetro?

```
calculator.xProjection(null, new Point(12, 13));
```

Receberemos uma `NullPointerException`, é claro.

Podemos consertar isso? Poderíamos criar um novo tipo de exceção e lançá-lo:

```
public class MetricsCalculator
{
    public double xProjection(Point p1, Point p2) {
        if (p1 == null || p2 == null) {
```

```
        throw new InvalidArgumentException(
            "Invalid argument for
MetricsCalculator.xProjection");
    }
    return (p2.x - p1.x) * 1.5;
}
}
```

Ficou melhor? Pode ser um pouco melhor do que uma exceção de ponteiro `null`, mas lembre-se de que temos de definir um tratador para `InvalidArgumentException`. O que ele deve fazer? Há algum procedimento bom?

Há uma alternativa. Poderíamos usar uma série de confirmações:

```
public class MetricsCalculator
{
    public double xProjection(Point p1, Point p2) {
        assert p1 != null : "p1 não pode ser nulo";
        assert p2 != null : "p2 não pode ser nulo";
        return (p2.x - p1.x) * 1.5;
    }
}
```

É uma boa informação, mas não resolve o problema. Se alguém passar `null`, ainda teremos um erro em tempo de execução.

Na maioria das linguagens de programação não há uma boa forma de lidar com um valor nulo passado acidentalmente para um chamador. Como aqui este é o caso, a abordagem lógica seria proibir, por padrão, a passagem de `null`. Ao fazer isso, você pode programar com o conhecimento de que um `null` numa lista de parâmetros é sinal de problema, e acabar com mais alguns erros por descuido.

Conclusão

Um código limpo é legível, mas também precisa ser robusto. Esses objetivos não são conflitantes. Podemos criar programas limpos e robustos se enxergarmos o tratamento de erro como uma preocupação à parte, algo que seja visível independentemente de nossa lógica principal. Na medida em que somos capazes de fazer isso, podemos pensar nisso de forma independente e dar um grande passo na capacidade de manutenção de nosso código.

Bibliografia

[Martin]: Agile Software Development: Principles, Patterns, and Practices, Robert C. Martin, Prentice Hall, 2002.

8. Limite os softwares que

8

Limites

por James Grenning



Raramente controlamos todos os softwares em nossos sistemas. De vez em quando compramos pacotes de outros fabricantes ou usamos códigos livres, ou dependemos de equipes em nossa própria empresa para construir componentes ou subsistemas para nós. De algum modo, devemos integrar, de forma limpa, esse código externo ao nosso. Neste capítulo veremos as práticas e técnicas para manter limpos os limites de nosso software.

O uso de códigos de terceiros

Há uma tensão natural entre o fornecedor de uma interface e seu usuário. Os fornecedores de pacotes e frameworks de outros fabricantes visam a uma maior aplicabilidade de modo que possam trabalhar com diversos ambientes e atender a um público maior. Já os usuários desejam uma interface voltada para suas próprias necessidades. Essa tensão pode causar problemas nos limites de nossos sistemas.

Tomemos o `java.util.Map` como exemplo. Como pode ver na Figura 8.1, os Maps tem uma interface bastante ampla com diversas capacidades. Certamente esse poder e flexibilidade são úteis, mas também pode ser uma desvantagem. Por exemplo, nosso aplicativo pode construir um Map e passá-lo adiante. Nossa objetivo talvez seja que nenhum dos recipientes de nosso Map não exclua nada do Map. Mas logo no início da lista está o método `clear()`. Qualquer usuário do Map tem o poder de apagá-lo. Ou talvez, segundo a convenção que adotamos, o Map pode armazenar apenas certos tipos de objetos, mas não é certo que ele restrinja os tipos que são adicionados a ele. Qualquer usuário determinado pode adicionar itens de qualquer tipo a qualquer Map.

Se nosso aplicativo precisa de um Map de Sensors, você pode se deparar com um Sensors assim:

- `clear() void - Map`
- `containsKey(Object key) boolean - Map`
- `containsValue(Object value) boolean - Map`
- `entrySet() Set - Map`
- `equals(Object o) boolean - Map`
- `get(Object key) Object - Map`
- `getClass() Class<? extends Object> - Object`
- `hashCode() int - Map`
- `isEmpty() boolean - Map`
- `keySet() Set - Map`
- `notify() void - Object`
- `notifyAll() void - Object`
- `put(Object key, Object value) Object - Map`
- `putAll(Map t) void - Map`
- `remove(Object key) Object - Map`
- `size() int - Map`
- `toString() String - Object`
- `values() Collection - Map`
- `wait() void - Object`
- `wait(long timeout) void - Object`
- `wait(long timeout, int nanos) void - Object`

Figura 8.1: Métodos do Map

Se nosso aplicativo precisar de um Map de sensors, você talvez encontre sensors assim:

```
Map sensors = new HashMap();
```

E, quando alguma outra parte do código precisa acessar o Sensor, você vê isso:

```
Sensor s = (Sensor)sensors.get(sensorId);
```

Não vemos isso apenas uma vez, mas várias ao longo do código. O cliente deste código fica com a responsabilidade de obter um `Object` do `Map` e atribuí-lo o tipo certo. Apesar de funcionar, não é um código limpo. Ademais, esse código não explica muito bem o que ele faz. Pode-se melhorar consideravelmente sua legibilidade com o uso de tipos genéricos, como mostra abaixo:

```
Map<Sensor> sensors = new HashMap<Sensor>();  
...  
Sensor s = sensors.get(sensorId);
```

Entretanto, isso não resolve o problema de que `Map<sensor>` oferece mais capacidade do que precisamos ou queremos.

Passar adiante pelo sistema uma instância de `Map<Sensor>` significa que haverá vários lugares para mexer se a interface para o `Map` mudar. Você talvez pense que uma mudança seja pouco provável, mas lembre-se de que houve uma quando o suporte a genéricos foi adicionado no Java 5. De fato, já vimos sistemas que impedem o uso de genéricos devido à magnitude das alterações necessárias para manter o uso abrangente dos `Maps`.

Abaixo está uma forma limpa de usar o `Map`. Nenhum usuário do `Sensors` se importaria se um pouco de genéricos for usado ou não. Essa escolha se tornou (e sempre deve ser) um detalhe da implementação.

```
public class Sensors {  
    private Map sensors = new HashMap();  
  
    public Sensor getById(String id) {  
        return (Sensor) sensors.get(id);  
    }  
    //codigo  
}
```

A interface no limite (`Map`) está oculta. É possível alterá-la causando muito pouco impacto no resto do aplicativo. O uso de tipos genéricos não é mais uma questão tão problemática assim, pois o gerenciamento de declarações e de tipos é feito dentro da classe `Sensors`.

Essa interface também foi personalizada para satisfazer as necessidades do aplicativo. Seu resultado é um código mais fácil de se entender e mais difícil de ser utilizado incorretamente. A classe `Sensors` pode forçar regras de modelo e de negócios.

Não estamos sugerindo que cada uso do `Map` seja encapsulado dessa forma. Mas lhe aconselhando para não passar os `Maps` (ou qualquer outra interface num limite) por todo o sistema. Se usar uma interface, como a `Map`, no limite, a mantenha numa classe ou próxima a uma família de classes em que ela possa ser usada. Evite retorná-la ou aceitá-la como parâmetro em APIs públicas.

Explorando e aprendendo sobre limites

Códigos de terceiros nos ajudam a obter mais funcionalidade em menos tempo. Por onde começar quando desejamos utilizar pacotes de terceiros? Não é tarefa nossa testá-los, mas pode ser melhor para nós criar testes para os códigos externos que formos usar.

Suponha que não esteja claro como usar uma biblioteca de terceiros. Podemos gastar um dia ou dois (até mais) lendo a documentação e decidindo como vamos usá-la. Então, escreveríamos

nosso código para usar o código externo e vemos se ele é ou não o que achávamos. Não ficaríamos surpresos de acabar em longas sessões de depuração tentando descobrir se os bugs que encontramos são do nosso código ou no deles.

Entender códigos de terceiros é difícil. Integrá-lo ao seu também é. Fazer ambos ao mesmo tempo dobra a dificuldade. E se adotássemos uma outra abordagem? Em vez de experimentar e tentar o novo código, poderíamos criar testes para explorar nosso conhecimento sobre ele. Jim Newkirk chama isso de *testes de aprendizagem*¹.

Nesses testes, chamamos a API do código externo como o faríamos ao usá-la em nosso aplicativo.

Basicamente estariamos controlando os experimentos que verificam nosso conhecimento daquela API.

O teste se focaliza no que desejamos saber sobre a API.

Aprendendo sobre log4j

Digamos que queremos usar o pacote log4j do Apache em vez de nosso próprio gravador de registro interno.

Baixaríamos o pacote e então abriríamos a página de documentação inicial. Sem ler muito, criamos nosso primeiro caso de teste, esperando que seja impresso “oi” no console.

```
@Test
public void testLogCreate() {
    Logger logger = Logger.getLogger("MyLogger");
    logger.info("oi");
}
```

Quando o executamos, o registrador (logger) produz um erro o qual nos diz que precisamos de algo chamado Appender. Após ler um pouco mais, descobrimos que existe um ConsoleAppender. Então, criamos um ConsoleAppender e vemos se desvendamos os segredos de registro no console.

```
@Test
public void testLogAddAppender() {
    Logger logger = Logger.getLogger("MyLogger");
    ConsoleAppender appender = new ConsoleAppender();
    logger.addAppender(appender);
    logger.info("oi");
}
```

Desta vez, descobrimos que o Appender não possui fluxo de saída. Estranho... Parecia lógico ter um. Depois de buscar ajuda no Google, tentamos o seguinte:

```
@Test
public void testLogAddAppender() {
    Logger logger = Logger.getLogger("MyLogger");
    logger.removeAllAppenders();
    logger.addAppender(new ConsoleAppender(
        new PatternLayout("%p %t %m%n"),
        ConsoleAppender.SYSTEM_OUT));
    logger.info("oi");
}
```