1

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 criarão 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.*

1. [Beck07].

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).

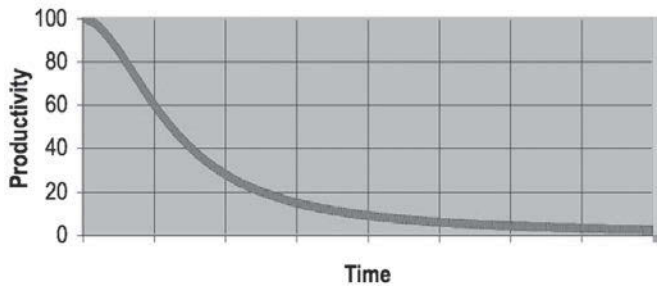


Figura 1.1
Produtividade v. tempo

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 Dilbert[*1], 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 demais e não teriam tempo para lavar as mãos entre um paciente e outro.

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 deselegância. 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.

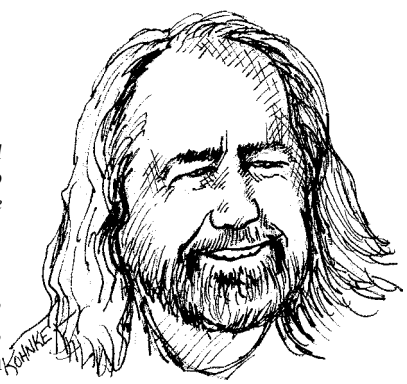
3. http://www.pragmaticprogrammer.com/booksellers/2004-12.html

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.*

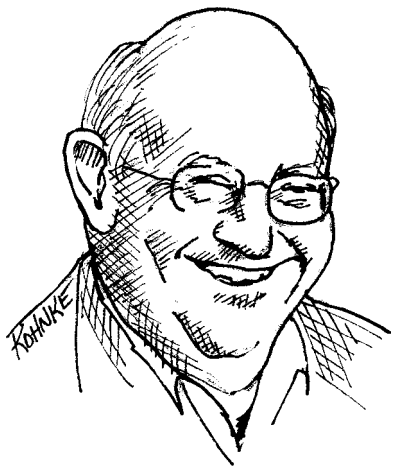


Grady fala de alguns dos mesmos pontos que Bjarne, voltando-se mais para questão da legibilidade. Eu, particularmente, gosto desse ponto de vista de que ler um código limpo deve ser como ler uma prosa bem escrita. Pense num livro muito bom que você já leu. Lembre-se de como as palavras eram substituídas 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 "explícito". 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



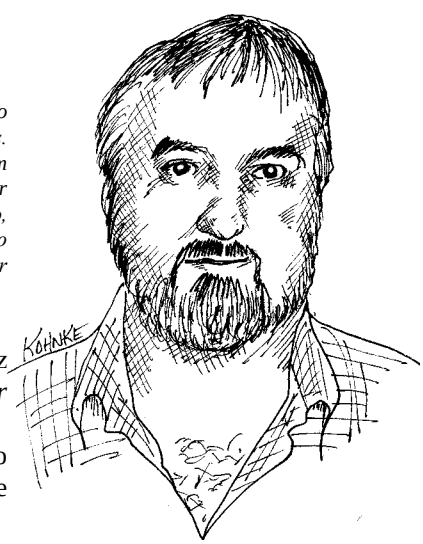
> *Além de seu criador, um desenvolvedor pode ler e melhorar um código limpo. Ele tem testes de unidade e de aceitação, nomes significativos; ele oferece apenas uma maneira, e não várias, de se fazer uma tarefa; possui poucas dependências, as quais são explicitamente declaradas e oferecem um API mínimo 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 quão 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 melhoras, você voltará ao início, ou seja, apreciando o código deixado para você por alguém que se importa bastante com essa tarefa.*

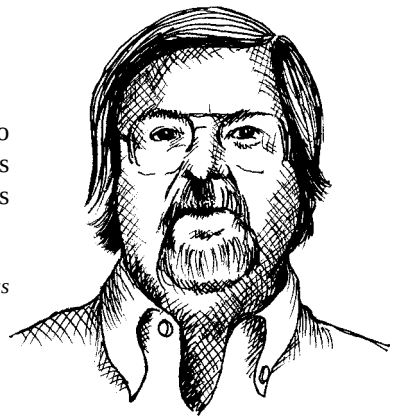
One word: care. É esse o assunto deste livro. Talvez um subtítulo apropriado seria *Como se importar com o código.*

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

4. [Knuth92].

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#*

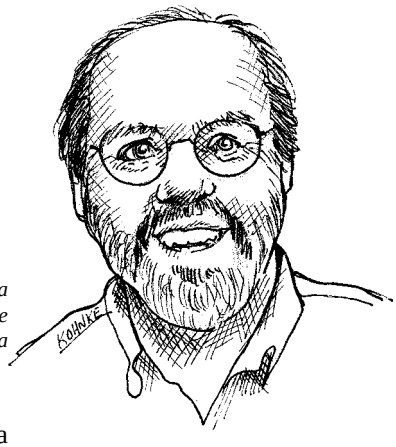


Ron iniciou sua carreira de programador em Fortran, no Strategic Air Command, e criou códigos em quase todas as linguagens e máquinas. Vale a pena considerar suas palavras:

> *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.*
> *Posso 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 início 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

5. Essa frase foi adaptada da mensagem de despedida de Robert Stephenson Smyth Baden-Powell para os Escoteiros: "Experimente e deixe este mundo um pouco melhor de como você o achou..."

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.