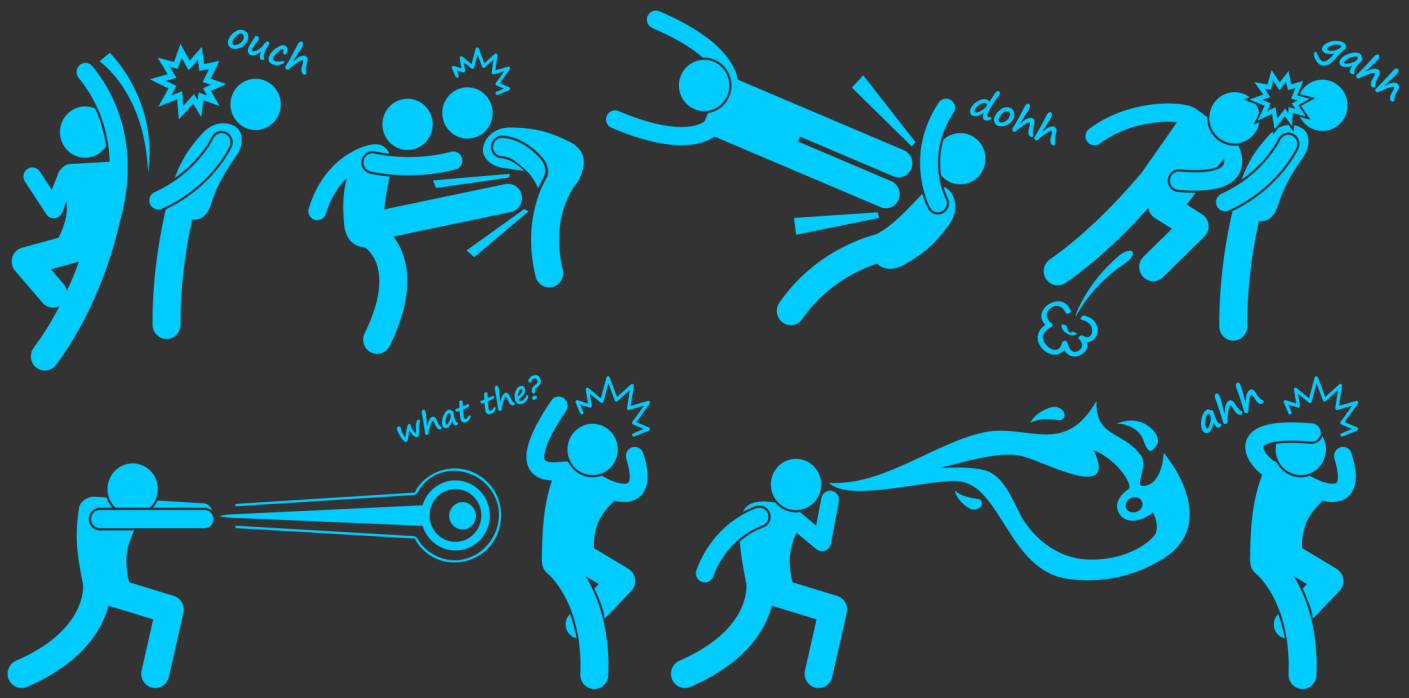

Programador Profissional

Márcio Torres

Técnicas e Práticas de Programação

Programador Profissional

Técnicas e Práticas de Codificação

Márcio Torres

Esse livro está à venda em http://leanpub.com/progpro

Essa versão foi publicada em 2021-04-15

Tweet Sobre Esse Livro!

Por favor ajude Márcio Torres a divulgar esse livro no Twitter!

O tweet sugerido para esse livro é:

Programe like a pro! Técnicas e Práticas de Programação
https://leanpub.com/o-programador-profissional-tecnicas-praticas-codificacao/ por @leanpub

Eu queria ser um mecânico, como meu pai. Mas ele disse que eu não sujaria minhas mãos de graxa. Então, dedico esse livro a ele, que sujou muito as mãos de graxa para que eu tivesse o conhecimento necessário para escrever um livro como este.

Conteúdo

Prefácio . . . . . **i**

Introdução . . . . . **ii**
Abordagem ii
Códigos dos exemplos ii
Organização dos capítulos iii
Para quem é este livro iii
Para quem não é este livro iv
Convenções iv
Sobre mim vi

Capítulo `000` – O Programador Profissional **1**
De amador a profissional 1
Sobre o livro 3
Existem diferenças entre programador e desenvolvedor? 4
O preço do amanhã está na habilidade de projetar 5
A palavra de ordem é **DI-VI-DIR** 5
O impacto na Qualidade de Software 6
Pedaços de mau código 6
Técnicas, Práticas, Princípios, Padrões e Bruxarias 14
Existe uma diferença entre conhecer o caminho e percorrer o caminho 15
O que vem a seguir 15

Prefácio

Em 2010 eu comecei a ministrar uma disciplina voltada ao projeto de aplicações. Essa disciplina era parte do curso superior de Tecnologia em Análise e Desenvolvimento de Sistemas ofertado pelo Instituto Federal de Educação, Ciência e Tecnologia Campus Rio Grande. Eu poderia dizer que foi uma disciplina feita para mim, já que meu trabalho na área, desde bastante tempo, sempre envolveu tomar decisões que impactavam diretamente na qualidade do código-fonte e do projeto em mais alto nível.

Nas primeiras aulas, eu sempre trazia anotações, formuladas a partir de um compilado dos livros recomendados, misturadas com experiências pessoais de trabalho e envoltas, sempre, em um contexto prático – *isto, já que os cursos de tecnologia são voltados para a aplicação prática.*

Meu principal desafio era o de conseguir uma abordagem didática. Então essas anotações foram desconstruídas e reconstruídas muitas vezes para serem *"ensináveis"* e *"aprendíveis"*. O resultado deste processo iterativo de refino do material é este livro didático, que preferi produzir no lugar dos *slides*.

Não foi fácil definir uma estrutura para este livro, além de decidir o quanto de conteúdo eu deveria colocar em cada tópico. Além disso, quanto mais progresso eu fazia, mais assuntos apareciam **– eis um livro sem fim**.

Sem mais delongas, este livro não foi escrito inicialmente para ser publicado. Ele foi escrito para ensinar em sala de aula. Porém, com o apoio de amigos, colegas, alunos e da família, aqui está ele, disponibilizado para o público geral. Se esta obra te for útil, mesmo que um pouco, já se pagou o trabalho para escrevê-la.

Introdução

Se eu vi mais longe, foi por estar sobre ombros de gigantes.

– Isaac Newton

Se eu fosse resumir o objetivo deste livro em uma linha ele seria:

Como construir software de qualidade dando atenção especial ao código-fonte..

Aprendizes e iniciantes na área da programação, ou desenvolvimento de software, como preferir, podem usar este livro como referência para o exercício da profissão.

Nesta obra há uma compilação de conhecimentos e habilidades necessárias para te ajudar a fazer boas decisões quando estás programando. O livro todo é sobre programar melhor.

Contudo, é importante dizer que de jeito nenhum se pretende esgotar o tema. Grande parte das ideias presentes aqui não vieram totalmente da minha cabeça. Ao contrário, as ideias são baseadas em grandes obras, as referências na área, escritas por excelentes profissionais, como Martin Fowler, Kent Beck, Joshua Bloch, entre outros.

Abordagem

Tenha em mente que este livro foi escrito a partir de notas de aula usadas no ensino de tópicos avançados de programação, tais como: técnicas, princípios e padrões de projeto de softwares, práticas específicas, etc. Esta origem deu ao livro as seguintes características:

- **Dedicado ao ensino**: foi projetado para ser didático, *"digerível"*, simples – mesmo que explique conceitos complexos;
- **Focado na prática**: foi pensando para o exercício do ofício, para trabalhar mesmo, e por isso é povoado de estudos de caso, exemplos e muitos códigos;
- **No nosso idioma e cultura**: não foi escrito para ser uma obra erudita e sim para ser simples de ler, acessível às pessoas que falam e/ou entendem português – com um leve sotaque gaúcho;

Códigos dos exemplos

A maior parte dos códigos está disponível na linguagem de programação Java. Ela é a linguagem mais usada na grande maioria dos livros de técnicas, práticas e padrões, isto é, sempre que a linguagem não é o foco do livro. Java também é bastante usada pelas faculdades e escolas técnicas, dado o suporte avançado aos conceitos de programação orientada a objetos e alinhamento com linguagens de modelagem visual, como a UML.

De fato, 97.312% do conteúdo deste livro é implementável em qualquer linguagem de programação, com poucos ajustes. O foco nunca é a linguagem, senão as técnicas! Para complementar, sempre que cabível, existem códigos de exemplo em outras linguagens como: C, PHP, Ruby, etc.

Todos os códigos estão disponíveis *online* em um repositório no github.com[1] neste endereço: https://github.com/marciojrtorres/tecnicas-praticas-codificacao. Fique à vontade para baixá-los, adaptá-los, testá-los, *forká-los*, faça a festa com eles, *mis códigos son sus códigos*, é tudo *opensource*!

Organização dos capítulos

O livro está organizado tanto para cobrir a escrita de códigos novos como também a reescrita (refatoração) de códigos existentes. Portanto, resulta nos seguintes capítulos:

- **Capítulo 000 – O Programador Profissional**: oferece uma introdução ao tema central deste livro: códigos.
- **Capítulo 001 – Boas Práticas de Codificação**: apresenta as melhores práticas usadas pelos profissionais para escrever códigos de alta qualidade técnica.
- **Capítulo 010 – Técnicas Avançadas de Codificação**: mostra as técnicas aplicadas por programadores experientes para escrever códigos sofisticados e elegantes.
- **Capítulo 011 – Melhorando Códigos Existentes**: apresenta técnicas e práticas usadas para melhorar uma lógica e códigos que já existem (o famoso código legado).
- **Capítulo 100 – Cenas dos Próximos Capítulos**: faz uma análise do que foi visto e traz um resumo do que ficou para próximas publicações.

Para quem é este livro

Este é um livro destinado à Educação Profissional. Foi testado em sala de aula e usado nas disciplinas de Aspectos Avançados de Programação, Arquitetura e Projeto de Software e Tópicos Avançados, nos cursos de Análise e Desenvolvimento de Sistemas e Técnico em Informática para Internet, ambos do Instituto Federal de Educação, Ciência e Tecnologia do Rio Grande do Sul (IFRS).

É uma obra útil para estudantes que já passaram pelas disciplinas iniciais de seus cursos (como Lógica ou Introdução à Programação e Programação Orientada a Objetos) e que pretendem escrever códigos profissionalmente, para ingressar e obter êxito no mundo de trabalho.

Também pode atender programadores formados e mesmo aqueles que nunca fizeram um curso, mas que já trabalham na área e buscam um material técnico especializado que vá além do básico, com uma abordagem direta e aplicação prática.

Líderes Técnicos ou Gerentes de Projeto podem usar este livro para treinar suas equipes. Programadores experientes podem achar os tópicos muito básicos, afinal já passaram pelas armadilhas e intempéries que motivam estas práticas, isto é, já são profissionais. Talvez, ainda possam encontrar algumas coisinhas interessantes.

[1] http://www.github.com

Academicamente, este livro é adequado ao ensino em cursos voltados para a Educação Profissional, Técnica e Tecnológica, tais como: Técnico em Informática, Técnico em Informática para Internet, Tecnologia em Análise e Desenvolvimento de Sistemas, Tecnologia em Sistemas para Internet, além de cursos de formação inicial e continuada na área de desenvolvimento de sistemas.

Ah, e antes de terminar, este livro é para as estudantes, para elas, para as mulheres na tecnologia, as desenvolvedoras, as programadoras. A língua portuguesa não possui, de fato, um gênero neutro. Portanto, saiba que sempre que eu menciono "programador", na minha cabeça a imagem não é aquele estereótipo hacker da TV, é gente real, jovem, adulta, idosa, homens e mulheres, peles claras e escuras, cabelos curtos, longos, sem cabelo, enfim, é da superclasse abstrata humano que estou falando.

Para quem não é este livro

Não cobre construções básicas de programas. O livro não tem o objetivo de ensinar a programar do zero. Ao contrário, é um livro para ensinar a programar melhor. Se não sabes programar, se não passaste por uma disciplina introdutória à programação, é bem provável que não seja muito útil.

Não é sobre algoritmos e estruturas de dados. Embora algumas técnicas sejam relacionadas à performance, o livro não aborda técnicas de escrita de algoritmos para obter eficiência computacional.

Não é teórico. Não espere deste livro uma abordagem teórica. Ele foi escrito por um praticante para praticantes.

Convenções

A seguir algumas convenções a respeito da abordagem, tipografia e *layout*.

Acrônimos

Algumas palavras e nomes aparecem abreviados, usando siglas ou acrônimos. Na primeira vez que forem exibidos constará o nome completo e no restante do livro (salvo exceções) é usado o acrônimo. Por exemplo, Programação Orientada a Objetos é abreviada como POO.

Inglês

Este livro tem a proposta de proporcionar material técnico no nosso idioma. Entretanto, na área de desenvolvimento de *softwares*, o idioma inglês é predominante – nos códigos é inevitável. Por este motivo, termos e nomes amplamente conhecidos em inglês terão uma explicação em português, mas serão apresentados na sua forma original (em inglês). É importante te habituares, pois mesmo que pareça estranho alguém dizer *"dropa a tabela"* ou *upa o código*, é o modo como as pessoas falam no ambiente de trabalho – isso é importante em um livro voltado para a educação profissionalizante. Fique tranquilo se não sabes ler textos em inglês, pois as explicações estão no nosso idioma.

Códigos

Trechos de código são exibidos usando fonte de largura fixa.

Em meio ao texto eles aparecem como neste exemplo: "*... usamos o método* `BaseDados.salva(Documento):boolea` *para ...*". As assinaturas de métodos são apresentadas como `Classe.metodo(TipoArgumento1, ..., TipoArgumentoN):TipoRetorno` (semelhante às notações UML, caso conheças).

Trechos mais extensos de código são apresentados na forma a seguir:

Blocos de código-fonte aparecem com um título como este, com a aparência a seguir e uma URL logo abaixo para o código completo.

```
1  /*
2   * Os códigos são exibidos usando
3   * uma fonte de largura fixa,
4   * como neste bloco.
5   */
6  public class UmaClasseDeExemplo {
7    public String UmMetodoDeExemplo(int umParametroDeExemplo) {
8      return "Um retorno com " + umParametroDeExemplo;
9    }
10 // ...
11 // https://gist.github.com/marciojrtorres/6159546
```

Comentários seguido por reticências `// ...` significam código omitido. Às vezes o código inteiro é muito longo para ser colocado na listagem, então a parte irrelevante para o exemplo é omitida. Se quiseres ver o código inteiro visite o *link* exibido na última linha da listagem como um comentário.

Fortemente te sugiro que copies e faças experiências com os códigos. Use-os como um *toy project* (projeto para aprendizado). Vai te garantir um aprendizado mais efetivo.

Finalmente, todos os códigos seguem a convenção (ou tentam seguir) do Google para o formato de códigos Java. Podes ver mais em Google Java Style Guide em https://google.github.io/styleguide/javaguide.html.

Dicas, Avisos e Observações

O livro está populado com dicas. A maioria é relacionada às práticas dos programadores profissionais, o que eles fazem, o que não fazem, com o quê se preocupam, como se alimentam e reproduzem – esta parte é brincadeira :). O formato está a seguir:

As boas práticas aparecem assim.

Com um título direto e uma explicação mais detalhada.

As más práticas aparecem assim.

Servem para serem evitadas, não seguidas.

O capiroto está nos detalhes.

Caixas como essa apresentam os problemas escondidos nos detalhes.

Observações gerais.

Aparecem assim, é isso.

Milongas e devaneios.

Essas caixas servem para abrigar devaneios de um velho programador milongueiro dos pampas. Não contém, a rigor, conteúdo técnico, então podes ignorá-las sem problemas.

Sobre mim

Atualmente sou Professor no IFRS, atuo no curso Técnico em Informática para Internet e no curso superior em Tecnologia em Analise e Desenvolvimento de Sistemas.

Meus alunos perguntam: *"professor, tu trabalha ou só dá aula?"*

É a vida de quem ensina, ter que ouvir isso. Enfim, **eu trabalho** em sala de aula, sou um educador, e ainda participo de projetos internos e dou uns *pitacos* em sistemas alheios (atividade conhecida como consultoria :).

Anterior a isto, tenho uma história longa. Eu passei por vários marcos na linha do tempo da evolução da computação (é, eu sou velho). Para me conheceres melhor vou contar um pouco dessa história a seguir:

Eu nasci a dez mil anos atrás. Comecei programando na linguagem Basic, num CP500 da prológica. Não havia Internet naquela época. Se aprendia lendo revistas técnicas, transcrevendo códigos e fazendo experiências de tentativa-e-erro. Mais tarde comecei a desenvolver em dBase e depois Clipper, ambos sobre a plataforma MS-DOS. Joguei Prince of Persia, Wolfenstein e o primeiro DOOM - tinha que usar o DOS/4GW para liberar a memória estendida (mais que os 640KB usuais). Já montei meu próprio computador – quando ainda tinha que selecionar IRQ através de jumpers ou switches. Vivenciei a ascensão da interface gráfica - que bom, pois não aguentava mais ver caracteres em fósforo verde. Instalei o Windows 95 - trocando malditos 13 disquetes um-a-um. Meu computador tinha um Kit Multimídia da Creative - e uma placa de vídeo Voodoo. Migrei meus sistemas de Clipper para Visual Basic e mais tarde Delphi. Usei a Internet quando só existia HTML e "meia dúzia" de tags - sem CSS ou JavaScript. Acompanhei a ascensão da Internet e da Web. Presenciei o início do Linux, sua evolução e sua importância para a consolidação dos sistemas online - junto com Perl, Apache, MySQL, PHP e outras várias tecnologias e ferramentas pioneiras. Já instalei o Conectiva Linux e então conheci o que é uma linha de comando de verdade. Comecei a programar em Java a partir da versão 1.3 e Java Enteprise (EE) a partir do 1.4. Ainda lembro como sofri para entender o que era Orientação

a Objetos - velhos hábitos procedurais são difíceis de perder. Acompanhei a Googlificação e o crescimento do comércio eletrônico - e também o estouro da bolha da Internet.

Não tenho dúvidas que sou um programador que ensina ao mesmo tempo que um professor que programa.

Capítulo `000` – O Programador Profissional

Sobre o futuro...

A melhor maneira de prever o futuro é inventá-lo.

– Alan Kay

Este capítulo trata dos conceitos usados no restante do livro, incluindo uma introdução ao que os programadores profissionais fazem. Não é exatamente um capítulo técnico, mas ele contém uma boa parte do que eu gostaria que tivessem me dito quando estava começando e penso que ele pode te ajudar nesta caminhada de *Pequeno Padawan* a *Mestre Jedi*.

De amador a profissional

O termo *programador* é muito sobrecarregado atualmente. Antigamente era uma das mais conhecidas profissões na área da computação, mas hoje existem muitos termos novos, tais como: Gerente de Projeto, Desenvolvedor de Software, Desenvolvedor Back-end, Arquiteto de Software, Desenvolvedor Web, Desenvolvedor Front-end, Engenheiro de Software, Engenheiro de UX, Especialista em Devops, etc – *bah, a lista é longa*.

Primeiro, é preciso estabelecer uma noção do que é *ser programador*, suas competências e responsabilidades. Então, na evolução deste livro, vamos tentar entender como **programador** mais do que um profissional com a habilidade simples de escrever códigos. Com a evolução dos projetos de *software*, linguagens, plataformas, etc, o papel do programador, o **ser programador**, também passou por evoluções e recebeu novos significados.

Programar, hoje, é uma atividade coletiva e multidisciplinar. O programador de hoje trabalha em sistemas onde o código-fonte compartilhado com dezenas, às vezes centenas, de outros programdores. Trabalha com sistemas heterogêneos, escritos com mais de uma linguagem, e multiplataformas, que rodam sobre diferentes servidores e ambientes. São sistemas que, embora tenham longos ciclos de vida, são entregues em pequenas partes e em curtos intervalos. Resumindo, muito mudou. Programador **não é** mais aquele cara que ficava *encaixotado* na sua baia. Aquele que era especialista em uma determinada linguagem e praticamente proprietário de uma determinada parte do sistema.

Outra questão, bem clara, é que as empresas não precisam de alguém que programe para construir funcionalidades e resolver problemas pontuais. Em vez, e de fato, elas precisam de alguém que programe para **resolver uma família de problemas atuais e futuros**. Elas precisam do programador que projeta e escreve um código-fonte legível, fácil de entender, sustentável, com boa performance. Precisam de programadores para não um sistema, senão uma **família de subsistemas**, novos e existentes, que se intercomunicam, rodam em espaços e tempos diferentes.

Há bastante gente ingressando no mercado de desenvolvimento de *software*. Tem muita gente nova e capacitada. Mas, infelizmente, poucos evoluem e ocupam uma posição de destaque, ou até cargos mais atrativos e com melhores rendimentos. A palavra-chave aqui é **responsabilidades**. Será que alguém evolui profissionalmente por ter mais responsabilidades? ou por ter mais responsabilidades alguém evolui profissionalmente[2]? Bem, o fato é que os **profissionais** adquirem **responsabilidades** – *ou as responsabilidades os procuram.*

Não é difícil perceber a diferença entre programadores amadores e profissionais – na verdade, é até bem fácil. O profissional é aquele que realmente se importa, aquele que sempre planeja, aquele que sempre projeta – *tudo é friamente calculado.* O amador é aquele que faz o mínimo necessário, com pouco ou sem planejamento, sem fazer projeções, sem se importar com o futuro do *software* que está escrevendo – *nem com seu próprio.*

Como vais ver no decorrer deste livro, existem muitos exemplos e eles ajudam a explicar conceitos muito abstratos usando uma situação concreta. A seguir, vou apresentar 10 situações (que separei entre tantas outras) as quais colocam lado-a-lado o perfil de um programador profissional e um amador:

1. **O profissional escreve um código inteligível, sustentável, que seus colegas entendem**. Por outro lado, o amador não se preocupa com a legibilidade e codifica como se o código nunca mais fosse ser visto outra vez na vida – *nem por ele mesmo.*
2. **O profissional resolve não apenas o problema atual, mas também os problemas futuros da mesma família**. O amador, por sua vez, faz o mínimo, resolvendo apenas o problema particular.
3. **O profissional cria pedaços reutilizáveis de código, constrói bibliotecas e componentes de *software* os quais são incluídos e compartilhados entre projetos**. No entanto, o amador cria códigos monolíticos, imóveis, e quando precisa reutilizar códigos (lógica) entre projetos, usa o famoso `CTRL+C` e `CTRL+V`, espalhando duplicatas por toda a base de código – *pior, por todas as bases de código.*
4. **O profissional conversa com seus colegas, difunde o conhecimento, troca experiências e usa o vocabulário técnico adequado para passar suas ideias**. Por outro lado, o amador evita se comunicar, não usa o vocabulário compartilhado, alheio, não se importa com princípios, nem padrões ou técnicas de projeto.
5. **O profissional conhece especificidades da linguagem em que está programando, como detalhes da sintaxe e as bibliotecas disponíveis**. O amador, no entanto, resolve tudo usando os construtos mais básicos e primitivos da ferramenta, além de frequentemente escrever um código que resolve um problema que já foi resolvido por alguma biblioteca ou um companheiro, pois julga ser sempre melhor à sua maneira – *sofre do efeito Dunning-Kruger*[3].
6. **O profissional usa a ferramenta certa para o problema certo**. O amador, por sua vez, tenta resolver todos os problemas de todos os tipos sempre com a mesma abordagem[4], olha tudo sob o mesmo ângulo.
7. **O profissional acredita na propriedade coletiva de código, aceita e sugere que seus colegas usem e alterem seu código, sugere e considera sugestões da equipe**. Por outro

[2] Paradoxo Tostines! Se não sabes o que é, experimente o seguinte artigo (por sua conta e risco): http://desciclopedia.org/wiki/Tostines.

[3] é como é chamado o efeito em que pessoas com pouco conhecimento sobre algo a acreditam saber muito mais que outros mais preparados. Mais em https://pt.wikipedia.org/wiki/Efeito_Dunning%E2%80%93Kruger.

[4] Lei do Instrumento: *se tudo o que tens é um martelo, então tudo te parece um prego.*

lado, o amador não quer que *"mexam"* no seu código e muito menos aceita *"mexer"* no código de terceiros, reproduzindo bordões como *"não toquem nesse código que ele é meu"* e/ou *"esse código não fui eu que fiz"*.

8. **O profissional lê livros técnicos, *blogs* de especialistas e gurus; está sempre atento às novidades**. O amador, bem, não se importa em ler. Quando o amador precisa de uma ideia, ele usa o Google, abrindo primeiro *link* que aparece[5] (geralmente um do *Stack Overflow*).
9. **O profissional testa seu código, projeta as entradas e saídas possíveis e se antecipa aos erros do cliente**. O amador testa apenas com a especificação que recebeu (se testar), situações não pensadas pela equipe de análise estão fadadas a sorte – *ou azar*.
10. **O profissional sabe usar ferramentas de desenvolvimento, conhece teclas de atalho, comandos do terminal e automatiza suas tarefas mais comuns**. O amador *"chafurda"* os menus, ignora comandos do console e reproduz repetitivamente e manualmente suas tarefas.

Sobre o livro

O objetivo deste livro é **oferecer os conhecimentos e habilidades para programar profissionalmente**. É um compilado de anos de experiência em programação com anos de sala de aula ensinando como programar de forma efetiva e eficaz.

Deste ponto em diante tu vais encontrar dicas do tipo *"considere"* e *"evite"*, apresentadas nos seguintes formatos:

Programadores Profissionais leem material técnico especializado.

Eles buscam a aquisição e melhoria de habilidades.

Guardar o conhecimento para si é uma prática amadora.

Programadores Profissionais compartilham ideias e sabem que evoluem mais rápido coletivamente do que individualmente.

Leva tempo para aprender a fazer boas decisões.

Todas essas dicas foram colecionadas a partir de anos de trabalho, observação, leitura e compartilhamento de experiências. São bons e maus exemplos que todo programador iniciante gostaria de conhecer para desenvolver uma carreira profícua na área.

[5] Conhecido como Desenvolvimento Orientado ao Google (Google Oriented Development – GOD).

Existem diferenças entre programador e desenvolvedor?

É uma dúvida muito comum (e polêmica). Existem muitos argumentos sobre o que é (e o que faz) um Programador e um Desenvolvedor de Software[6]. Não é minha intenção trazer uma opinião unilateral e impor uma definição. Portanto, é necessário ficar claro que existem duas linhas principais de pensamento a respeito:

1: Uma linha de pensamento é de que Programador e Desenvolvedor são sinônimos, que uma definição ou outra pode ser usada para expressar uma *"pessoa que escreve códigos que computadores executam"*.

2: A outra linha de pensamento é que Programador e Desenvolvedor são profissões diferentes. O argumento é que Programadores escrevem códigos seguindo uma especificação, enquanto Desenvolvedores especificam, projetam e escrevem programas.

Honestamente, tenho dúvida sobre qual é a melhor definição. Contudo, confesso que na minha carreira vi claramente perfis de profissionais que se encaixam como *"só codificares"*, que escreviam a partir de uma especificação sem questioná-la ou projetá-la, bem como outros que planejavam, projetavam, codificavam e testavam, exercendo atividades em várias etapas do (longo) processo de desenvolvimento de *softwares*.

O tema é muito polêmico. Minha sugestão é que ouças e leias outras opiniões sobre Programador/Programação e Desenvolvedor/Desenvolvimento (sem falar de Engenheiros de Software, Cientistas da Computação, Engenheiros de Computação, etc). Por exemplo, podes fazer uma pesquisa a respeito: https://www.google.com.br/search?q=programador+desenvolvedor – *deves achar absurdo um livro te jogar para o Google para esclarecer uma dúvida, mas não quero entrar nessa treta*!

Então? Sugiro que façamos assim: no decorrer deste livro usarei o termo **Programador**. Este livro é sobre escrever códigos de alta qualidade, que parece a exata competência de um programador – *lógico, na minha humilde opinião.*

Importante! Partiremos também do pressuposto que todo Programador planeja, todo Programador faz projeções a respeito do código, isto é, se pressupõe que **o Programador projeta e escreve programas**, para efeito neste livro.

Programadores profissionais projetam.

Sempre que dedicas um minuto a mais para nomeares uma variável, escolheres a quantidade de parâmetros de um método e pensares sobre o tipo de saída esperada, bem, isso é projetar! Me parece que, de fato, programadores planejam, programadores projetam – *e codificam, obrivamente.*

[6] Bem, pelo menos sabemos que não é *"o rapaz ou a guria dos computador que arruma a impressora"*.

O preço do amanhã está na habilidade de projetar

Os programas, ou sistemas, costumam ter uma vida *looooonga*. Eles recebem adições e alterações de funcionalidades durante muitos anos. Dito isso, pense que os programas precisam de um planejamento, para crescer de forma *ordenada*. Assim como as cidades, eles também podem sofrer de *crescimento desordenado*. Com essa analogia acredito que consigas imaginar os sintomas e o resultado de um crescimento sem controle, certo? Logo, os programas precisam de um *"Plano Diretor"*.

Sendo direto, os programas precisam de um planejamento claro, consistente e com o olho no futuro, para permitir que cada mudança não se torne um evento traumático. Isto é, o planejamento existe para permitir que o programa seja fácil de alterar sem perder sua eficiência, estabilidade e outros atributos qualitativos. Em outras palavras, tu deves projetar e escrever códigos com boa qualidade técnica, para que possas alterá-los no futuro sem medo de que ele vá quebrar ao tocar (esta, aliás, é a definição de sistemas robustos, em oposição aos sistemas frágeis).

Programas implementados com pouco (ou nenhum) projeto anterior, tendem a ser instáveis e frágeis. Estes tipos de sistema rejeitam alterações e são difíceis de lidar. Programas que não foram projetados dificultam a submissão de modificações, sem precisar que sejam feitas alterações em cascata e, pior, sem introduzir três erros ao tentar corrigir um – *é a Hidra, corte uma cabeça e duas nascem.*

O segredo do sucesso das grandes empresas que desenvolvem *software* de qualidade é fazer boas projeções. Um bom projeto (no sentido de projetar, desenhar, *design* em inglês) garante uma vida mais saudável ao programa, com mais estabilidade, eficiência e, principalmente, suscetibilidade às alterações (característica qualitativa conhecida como flexibilidade).

Projetar programas refere-se a projetar qualquer "porção" dele. Ou seja, qualquer parte do programa pode ser codificada sem ou com projeções. Ou seja, tu podes codificar apenas para cumprir a especificação, sem te preocupar com o futuro, ou podes dar um minuto a mais de raciocínio quando escreveres classes, atributos, funções, métodos, parâmetros, estruturas de dados e quaisquer algoritmos. Enquadre isto como **investimeno** e não **perda** de tempo.

A palavra de ordem é DI-VI-DIR

Projetar envolve fazer escolhas difíceis que influenciam o futuro do programa. A boa notícia é que existem princípios, práticas e padrões que ajudam a fazer estas escolhas. Mas, se reduzires todos os conselhos em um só, qual seria? Ele seria **DI-VI-DIR**!

Dividir um programa em pequenas partes traz várias vantagens. Por exemplo, permite que as partes sejam reutilizadas, testadas e combinadas para criar partes maiores. Porém, dividir também traz uma desvantagem, que é o aumento de indireção, ou seja, uma funcionalidade vai depender de outra, que vai depender de outra e assim por diante.

O principal pecado dos sistemas mal projetados está em criar programas monolíticos (uma grande peça única). Considere como exemplo concreto um método (ou função), que pode ser projetado ou não. Projetar métodos implica em raciocinar sobre ele, fazendo projeções a respeito da inteligibilidade, estabilidade, flexibilidade e reuso. É bem comum, quando se projeta

métodos, perceber que ele precisa ser quebrado em métodos menores, cada um resolvendo um subproblema. Alguém que codifica sem projetar tende a resolver tudo de forma monolítica, resultando em poucas classes e métodos, todos com muito código – ver God Class[7].

Escrever monoblocos é uma prática amadora.

Programadores Profissionais sabem que dividir é fundamental pois cada parte pode ser testada individualmente. A qualidade das partes implica na qualidade do todo.

Confesso que conheço muitas pessoas relutantes em dividir o programa. Desde alunos, que estão iniciando, até colegas de trabalho, que programam *desde que tudo isto aqui era mato*. Todos têm algum argumento falacioso para não separar o programa em partes menores. Bem, é preciso ter em mente que qualquer atividade de engenharia, seja civil, mecânica, etc, trabalha com a ideia de construir em partes e juntá-las para construir partes maiores até ter o produto completo. Embora especialmente diferente, a Engenharia de Software, em essência, tende ao mesmo princípio das outras engenharias, sendo que a única diferença é que as peças são *lógicas* e não *físicas*.

O impacto na Qualidade de Software

Todo programa faz o que tem que fazer (embora existam controvérsias :). Cumprir suas funcionalidades sem erros é a obrigação de um programa. É a principal métrica de qualidade percebida pelo usuário final. Entretanto, existem outras métricas de qualidade. Por exemplo, aquelas que aparecem no momento em que o programa precisa passar por alterações.

A Garantia da Qualidade do Software é uma disciplina importante na Engenharia do Software. Existem várias preocupações, muito além da qualidade do código. Entretanto, o código-fonte tem influência direta em tudo, ou seja, escrever código de boa qualidade técnica é um caminho para construir *softwares* de melhor qualidade.

Programadores Profissionais sabem da importância do código na qualidade final do *software*.

Mesmo as pessoas que não trabalham com o código de perto – como Gerentes de Projeto, Analistas de Sistemas e Administradores de Bases de Dados – sabem da importância e prezam pela qualidade do código-fonte.

Pedaços de mau código

Então, se a qualidade do código é tão importante para a qualidade final de um sistema, como podes controlar isso? É esperado que os iniciantes tenham dificuldade de diferenciar um bom de um mau código. Eu sei disso porque (1) já fui iniciante e (2) sou professor atendo alunos nos cursos técnico e superior, que são iniciantes. A métrica básica para um bom código costuma ser `funciona => bom` e `não funciona => ruim`, infelizmente.

[7] http://c2.com/cgi/wiki?GodClass

Contudo, com um pouco de experiência e prática, é fácil identificar um código com baixa qualidade técnica. Isto é possível usando métricas melhores do que simplesmente *funciona ou não funciona*. Não é incomum um programador experiente abrir seus programas velhos e pensar *"como foi que eu escrevi essa m*?!"*.

Programadores Profissionais julgam a qualidade técnica do código.

Eles conseguem diferenciar um código bem escrito de outros mais *relaxados*.

Como uma introdução à codificação com profissionalismo, eu vou apresentar, a seguir, dois exemplos de problemas comuns presentes nos códigos de baixa qualidade técnica. Importante: estes programas funcionam (nunca é para discutir o funcionamento, funcionar é o mínimo), mas têm os seguintes problemas:

O problema dos nomes

Um dos problemas comuns encontrado nos *"códigos da vida"* é a má escolha de nomes. Considere um encontro com o código a seguir:

Nomes que não dizem o que o código faz

```
package problema;

public class Util {

  public static String paraiso(String s) {
    return s.split("/")[2] + "-" + s.split("/")[1] + "-" + s.split("/")[0];
  }
}
// https://git.io/JOswt
// https://github.com/marciojrtorres/tecnicas-praticas-codificacao/blob/master/pr\
oblema-dos-nomes/src/problema/Util.java
```

A pergunta é: **o que esse método faz?** Estou no paraíso? Que ótimo! Sarcasmo a parte, claro que, com leitura, raciocínio e experimentos, é possível descobrir o que ele faz. No entanto, pense que todo código precisa de manutenção – *e a manutenção, a manutenção* – que tem um custo conferido pela seguinte fórmula:

```
custo_manutenção = custo_entender + custo_alterar + custo_testar + custo_implantar
```

Em geral, classes, métodos e variáveis com nomes obscuros, ou que não compartilham o significado com quem está lendo, acabam aumentando o custo para entender, logo para alterar e, portanto, aumentando o custo de manutenção. Custo é tempo, tempo é dinheiro. Maior custo significa perder dinheiro; resultado: **maus nomes custam dinheiro!**

Para um Programador, maior *custo* significa necessidade de mais tempo.

Para um Gerente de Projetos, maior custo significa menor lucro – e para um GP neurótico fã de GoT significa que cabeças vão rolar :/

Clarificando, este método converte uma data recebido como uma `String` e no formato português brasileiro `dd/mm/aaaa` para o formato ISO `aaaa-mm-dd`. O nome `paraiso(String):String` significa *data para o formato ISO*[8].

Absurdo? Não.

Tu deves estar pensando: *"Quem daria um nome maluco para uma função como essa?"* Se serve de alento, na versão anterior deste livro eu tinha usado o nome `cotoiso(String):String` com o sentido de *converter para ISO*, que já vi uma vez :/ (1) existem nomes muito piores (2) que são dados por programadores que escrevem códigos sem pensar muito, sem raciocinar para além do código que estão *codando* no momento. Eu digo isso, porque na hora em que se está escrevendo, os nomes parecem muito claros. Precisa, às vezes, de uma visão mais distanciada para notar ... "opa, não está claro como deveria :".

A maneira básica, tradicional, de descobrir o que um método ou função faz é: *chama* e observa a saída. Por exemplo:

Observando a saída

```
package problema;

public class Main {
  public static void main(String[] args) {
    System.out.println(Util.paraiso("19/08/2014")); // imprime 2014-08-19
  }
}
// https://git.io/JOGaA
// https://github.com/marciojrtorres/tecnicas-praticas-codificacao/blob/master/pr\
oblema-dos-nomes/src/problema/Main.java
```

Métodos (ou funções) que precisam ser executados para serem entendidos são exemplos de maus códigos.

Os métodos devem ser projetados de modo que sejam autoexplicativos.

É possível entender o método se ele for invocado e, então, observar a saída. No entanto, se fossem usados nomes melhores, tanto para a classe como para o método, não seriam necessárias

[8] Datas representadas no formato internacional ISO8601 são apresentadas como yyyy-mm-dd. É o formato neutro, usado por exemplo nos bancos de dados. Mais em https://pt.wikipedia.org/wiki/ISO_8601.

experiências para entendê-lo. Classes, métodos e variáveis devem ser projetados para serem entendidos em uma leitura, como uma oração no sentido textual. O objetivo é converter uma data do formato BR para o formato ISO, portanto, veja a mesma funcionalidade codificada de forma mais descritiva:

Projetando nomes melhores

```
package solucao;

public class Data {
  public static String deBRparaISO(String dataBR) {
    String[] partes = dataBR.split("/");
    String dia = partes[0];
    String mes = partes[1];
    String ano = partes[2];
    return String.format("%s-%s-%s", ano, mes, dia);
  }
}
// https://github.com/marciojrtorres/tecnicas-praticas-codificacao/blob/master/pr\
oblema-dos-nomes/src/solucao/Date.java
```

Programadores Profissionais escrevem métodos autoexplicativos.

Eles sabem que códigos são escritos para serem lidos.

Para alguns a diferença parece sutil e até irrelevante, mas veja a seguir a chamada do novo método:

Projetado para ser óbvio

```
package solucao;

public class Main {

  public static void main(String[] args) {
    // Em vez de: System.out.println(Util.paraiso("19/08/2014"));
    System.out.println(Data.deBRparaISO("19/08/2014"));
  }
}
// https://github.com/marciojrtorres/tecnicas-praticas-codificacao/blob/master/pr\
oblema-dos-nomes/src/solucao/Main.java
```

É mais expressivo chamar um método nomeado como `Data.deBRparaISO("19/08/2014")` do que `Util.paraiso("19/08/2014")`. A regra geral é que classes e métodos devem ser escritos para serem lidos, como uma redação.

A vantagem dos anglofalantes sobre o resto de nós, lusofalantes, hispanohablantes, etc.

Ocorre que para os anglofalantes, isto é, falantes nativos da língua inglesa, as próprias instruções da linguagens são expressões textuais naturais – a eles, se parecem menos como código. Se o mesmo código fosse escrito em inglês, seria `Date.fromBRtoISO(String):String`. Por outro lado, se as instruções fossem em português, teríamos uma `classe pública Data { ... estático público Cadeia deBRparaISO(Cadeia dataBR) {`. Estranho como nos acostumamos a escrever em outro idioma ao ponto do código na nossa língua parecer alienígena.

O problema das expressões condicionais confusas

Construir e projetar expressões condicionais parece, e geralmente é, uma tarefa trivial. No entanto, escrever condicionais exige atenção à certos detalhes que envolvem performance, clareza, concisão, entre outros cuidados – *daria para escrever um livro só sobre isso, acredite*. O pior caso é quando as expressões condicionais são mal utilizadas tecnicamente, como no exemplo a seguir:

Usando o else em vez do if

```
// se a pesquisa tem termos
if (pesq.getTermos().size() == 0) {
} else {
  sql += " WHERE descricao LIKE ?";
}
```

Este caso é um exemplo de ou imperícia ou negligência – *o segundo é pior*. Programadores Profissionais não escrevem códigos assim – *e nem mesmo os iniciantes mais atinados*. Claro que o `else` não é necessário e esse código poderia ser escrito assim:

Usando if's responsavelmente

```
// se a pesquisa tem termos
if (pesq.getTermos().size() > 0) {
  sql += " WHERE descricao LIKE ?";
}
```

O mesmo código poderia ser ainda mais expressivo, aplicando uma refatoração bastante comum, chamada *método consulta*. O código fica mais inteligível quando a consulta é realizada diretamente no objeto (em vez de codificada em uma expressão). Isto ajuda a dispensar os comentários explicativos. Veja o mesmo código refatorado no exemplo a seguir:

Usando um método consulta em vez de expressão

```
if (pesquisa.temTermos()) {
  sql += " WHERE descricao LIKE ?";
}
```

Programadores Profissionais projetam suas expressões condicionais.

O objetivo é que elas sejam claras, sucintas e tenham uma boa performance.

Existem outros exemplos de *más* utilizações de expressões condicionais, onde elas poderiam ser melhor escritas ou até eliminadas, como no código a seguir:

Saraivada de if's e else's

```
1  package problema;
2
3  public class Util {
4
5    public static int dias(int mes, int ano) {
6      if (mes == 1) {
7        return 31;
8      } else if (mes == 2) {
9        if (ano % 400 == 0) {
10         return 29;
11       } else {
12         if (ano % 4 == 0 && ano % 100 > 0) {
13           return 29;
14         } else {
15           return 28;
16         }
17       }
18     } else if (mes == 3) {
19       return 31;
20     } else if (mes == 4) {
21       return 30;
22     } else if (mes == 5) {
23       return 31;
24     } else if (mes == 6) {
25       return 30;
26     } else if (mes == 7) {
27       return 31;
```

O interessante nesse código é que ele funciona! Ele devolve exatamente a quantidade de dias dado um mês e ano. Entretanto, o código é pouco inteligível. Com certeza poderia ser melhor escrito, a começar pelo tamanho – o método tem mais de 40 linhas.

Escrever códigos com muitas linhas é uma prática amadora.

Programadores Profissionais conhecem a linguagem o suficiente para escrever códigos mais concisos. Além disso, optam por dividir grandes funcionalidades em partes menores. Eles devem sabem que classes, métodos e expressões pequenas têm mais valor.

O mesmo problema pode ser resolvido com o mínimo de *if's*. Porém, o curioso é que o mesmo problema resolvido com menos linhas (e menos código) ainda pode parecer estranho, como no código a seguir:

Pequeno, mas estranho

```
package problema;

public class Util2 {

  public static int dias(int mes, int ano) {
    if (mes == 1 || mes == 3 || mes == 5 || mes == 7 ||
        mes == 8 || mes == 10|| mes == 12)
      return 31;
    else if (mes == 2 && ((ano % 400 == 0)
        || (ano % 4 == 0 && ano % 100 > 0)))
      return 29;
    else if (mes == 2)
      return 28;
    else if (mes == 4 || mes == 6 ||
        mes == 9 || mes == 11)
      return 30;
    throw new IllegalArgumentException("mes invalido");
  }
}
// https://github.com/marciojrtorres/tecnicas-praticas-codificacao/blob/master/pr\
oblema-expressoes-condicionais/src/problema/DateUtil2.java
```

A moral da história é: métodos com poucas linhas (código concisos) não necessariamente são claros (inteligíveis). O código anterior serve para demonstrar que mesmo um código pequeno pode ser confuso. Neste exemplo em particular, algumas boas práticas foram ignoradas, como a convenção de sempre usar chaves `{}` nos *if's – mesmo naqueles que tenham apenas uma instrução.*

Se longo é ruim e curto é ruim também, o que eu faço?!

O segredo, *pequeno padawan*, é como tudo na vida, achar o equilíbrio ... o balanço na força!

A maioria das expressões condicionais podem ser escritas de modo mais simples com o uso de algumas técnicas de refatoração. Por exemplo, com a introdução de uma variável explicativa ou

de um método consulta (refatorações serão vistas adiante neste livro no Capítulo 11: Melhorando Códigos Existentes). Na prática, algumas expressões condicionais podem ser até eliminadas. O código a seguir mostra uma abordagem diferente, com o uso de métodos auxiliares e um *array* de inteiros servindo como um mapa `mês => dias` para eliminar os testes condicionais longos:

Código para ser lido

```
package solucao;

public class Data {
  private static final int[] diasMes = {0, // HACK (ou KLUGE) do índice nulo
          31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
  // mês:   1   2   3   4   5   6   7   8   9   10   11   12
  private static final int FEVEREIRO = 2;

  private static boolean anoBissexto(int ano) {
    return ano % 400 == 0 || (ano % 4 == 0 && ano % 100 > 0);
  }

  private static boolean mesInvalido(int mes) {
    return mes < 1 || mes > 12;
  }

  public static int dias(int mes, int ano) {
    if (mesInvalido(mes)) {
      throw new IllegalArgumentException("Mês não é válido");
    }
    if (anoBissexto(ano) && mes == FEVEREIRO) {
      return 29;
    }
    return diasMes[mes];
  }
}
// https://github.com/marciojrtorres/tecnicas-praticas-codificacao/blob/master/pr\
oblema-expressoes-condicionais/src/solucao/DateUtil.java
```

Programar é uma *"arte"* ou uma *"ciência"*? Os dois.

É possível construir uma mesma funcionalidade de muitos modos diferentes – *algumas muito criativas*. O código anterior ainda tem vários pontos questionáveis, como o *hack*[9] na declaração do *array*. Contudo, o método terminou com uma implementação textual inteligível – *até passar por um nova revisão (ver o Princípio do Bom Suficiente).*

Programadores Profissionais escrevem códigos textuais.

Eles escrevem código com início, meio e fim, tal como uma redação.

[9] *hacks*, também conhecidos como *kluges* ou *kludges*, são soluções rápidas, feias, mas criativas (e **questionáveis**). São *gambiarras* de alta qualidade realizadas por profissionais (o qué é isso? existem níveis de gambiarra? existem.). Uma explicação completa do termo pode ser encontrada no site *Jargon File* do Eric Raymond em http://www.catb.org/jargon/html/K/kluge.html.

Fazer gambiarras é uma prática amadora.

Os profissionais escrevem *hacks*, não gambiarras – *mas não diga que aprendeu isso neste livro :P*

A raiz de todo o mal

A maioria dos códigos difíceis de ler e entender acontecem ou por desconhecimento ou por negligência – *o primeiro motivo tem solução.* Todos os códigos podem ser esclarecidos com bons nomes e a introdução de código explicativo. Ademais, códigos escritos em pequenas partes são reutilizáveis e combináveis.

Regra de Ouro

`muito longo == ruim`, `curto == bom`, `muito curto = criptográfico` – *com algumas exceções aqui e ali ...*

Técnicas, Práticas, Princípios, Padrões e Bruxarias

Bom *software* é estável, flexível, confiável, fácil de alterar e introduzir novas funcionalidades, etc. No entanto, como se atinge este nível de qualidade?

É possível construir *software* de qualidade se fores um profissional disciplinado, se dominares a linguagem de programação e conheceres algumas técnicas, práticas, princípios e padrões, que são conhecidos e difundidos na comunidade de desenvolvimento de *software*.

Boas práticas nascem de bons hábitos. As boas práticas são frutos das ações disciplinadas, por exemplo, de estabelecer padronização para a escrita do código, seu formato e estrutura.

Técnicas são soluções especiais, sofisticadas, para problemas recorrentes. São normalmente aprendidas, ou descobertas, com o tempo. Isto é, são fruto da experiência. Algumas dessas técnicas são catalogadas e difundidas na comunidade de desenvolvimento de *software*.

Às vezes são detalhes óbvios, mas são eles que diferenciam o profissional do amador. Saber as práticas e técnicas é só o começo de uma carreira. A atividade de programar, de desenvolver *software*, está cheia de *mantras*, que vão além de escrever código. A profissão exige conhecimentos de planejamento, arquitetura e projeto de sistemas, utilização de princípios e padrões de projeto, realização de testes, depuração e medições – *na verdade, até pensei em introduzir tudo isso neste livro, mas ficaram para outros volumes, este aqui é dedicado à codificação.*

Opa, e as bruxarias?!

Já ia esquecendo. Considere e execute o seguinte código: `Integer a = 1; int b = 1; Integer c = 1; System.out.println(a == b); System.out.println(b == c); System.out.println(a == c);`. Tudo `true`, bem óbvio, certo? Agora invoque o *Expecto Patronum* e experimente um número maior, por exemplo `1000` _– e relate sua experiência. _Gostou?

Mais um: execute a instrução `System.out.println("abc" == "abc");`. Imprime `true`, como

esperado. Agora faça `System.out.println("abc".toUpperCase() == "abc".toUpperCase());`, `"ABC"` é igual a `"ABC"`?

Agora a saideira: é verdadeiro que `0.1 == 0.1`, é verdadeiro que `0.1 + 0.1 == 0.2`, será verdadeiro que `0.1 + 0.1 + 0.1 == 0.3`?

Existe uma diferença entre conhecer o caminho e percorrer o caminho ...

Mesmo que saibas inúmeras técnicas e práticas sofisticadas de programação, só vai fazer diferença se elas forem usadas efetivamente. Não é só para escrever pequenos experimentos. É para programar de verdade.

A dúvida comum é se somos ou não competentes para programar mais sofisticadamente. Não penses ou tentes ser um profissional, saibas que és um[10] – *pois se te pagam para programar.*

E lembres que maus hábitos são difíceis de perder. Por outro lado, novos e bons hábitos são difíceis de cultivar. Tudo é uma questão de persistência, de força de vontade mesmo. Ninguém é bom no que faz sem treinar.

O que vem a seguir

Após toda esta introdução, eu espero que estejas motivado. Programadores constroem coisas incríveis todos os dias. Obviamente, são os profissionais do presente, mas principalmente do futuro.

O restante deste livro está cheio de conteúdo para programadores – *de programador para programador.* Orgulhe-se dessa profissão e valorize-a[11] – *nós vamos dominar o mundo (risada maligna neste momento).*

[10] https://youtu.be/xkm9QJMbOYU
[11] https://youtu.be/nKIu9yen5nc