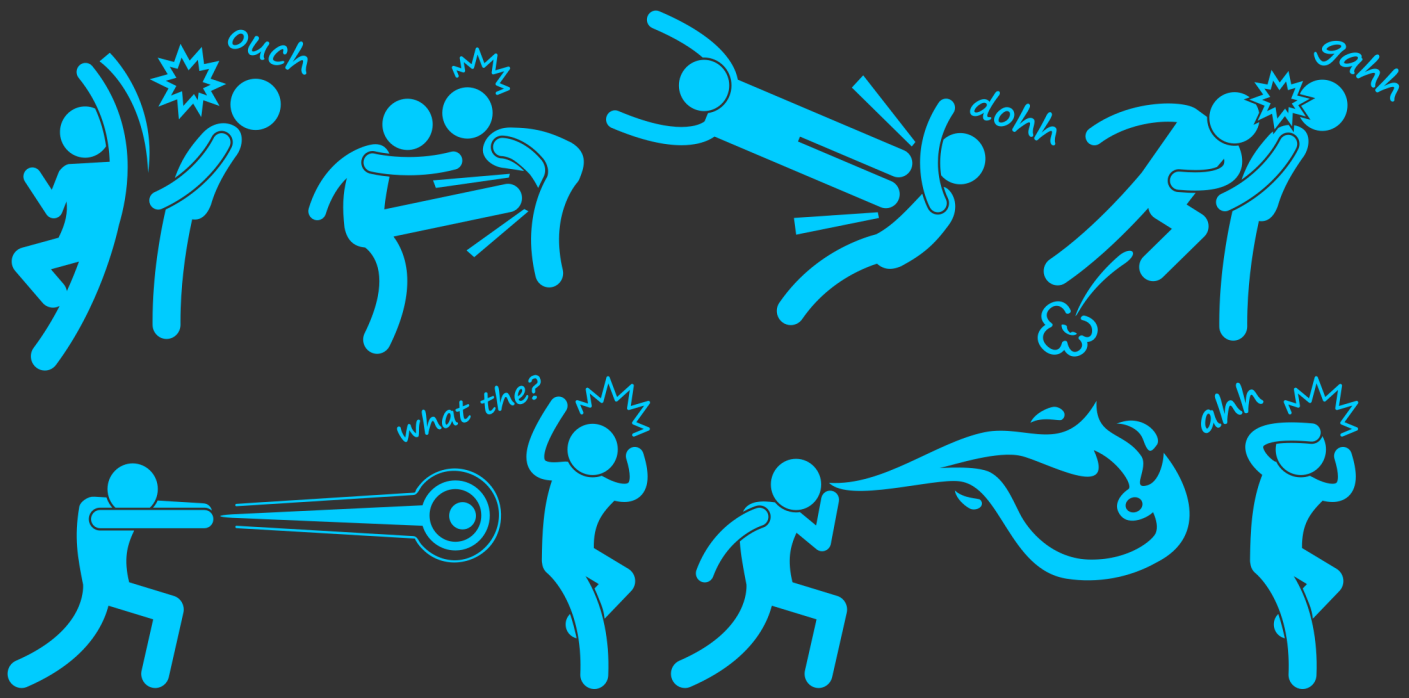


Programador Profissional

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

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

Capítulo 001 – Boas Práticas de Programação

Códigos que os outros possam entender

Uma diferença entre um programador inteligente e um programador profissional é que o profissional entende que a clareza é indispensável. Profissionais escrevem código que outros possam entender

– Robert Cecil Martin (Uncle Bob)

Como diferenciar um bom de um mau código além da métrica simples e direta *"ele funciona?"*? Por exemplo, será que consegues entender o objetivo de um método apenas olhando para o código durante um minuto? Se sim, ótimo, senão, então ele pode ser um código obscuro. Embora alguns códigos funcionem, podem não ser um exemplo de excelência técnica.

Códigos bem escritos são fáceis de serem lidos e entendidos. Com uma olhada rápida se descobre o objetivo, o que fazem e como fazem. Códigos obscuros e confusos trazem vários problemas e efeitos colaterais, destacando-se:

Redução da produtividade: leva mais tempo para alterar um código confuso e, conforme a base de código aumenta, acaba se tornando mais e mais difícil até tornar-se impossível de mantê-lo;

Facilidade de introduzir bugs: um código ruim pode leva o programador a fazer uma má interpretação ou obter um mau entendimento. Isso leva a facilidade de introduzir defeitos, ou seja, no ato de corrigir ou introduzir funcionalidades pode-se acabar incorrendo na introdução de novos *bugs*;

Dificuldade de eliminar bugs: os programadores podem levar um bom tempo tentando compreender um código ruim, para só então localizar o problema e corrigi-lo;

Excesso de trabalho: isso pode levar a situações como *"não vou mexer nesse código, passa para o -SeuNomeAqui-, ele que fez e só ele entende"*. Em outras palavras, escrever códigos feios e complexos vão te fazer trabalhar mais, pois se ninguém entender teu código ele vai acabar caindo sempre no teu colo `o_0`.

Declínio da popularidade e riscos a saúde: seus colegas de trabalho vão te odiar quando puserem a mão no teu código – e podem tentar contaminar teu café :)

Programadores Profissionais se preocupam em escrever códigos com alto nível de inteligibilidade.

Os profissionais conhecem as implicações e desdobramentos de um código confuso, por isso eles "doam" uns minutos a mais para deixá-lo óbvio, **sem interpretações dúbias e sem pontos questionáveis**. Bons códigos não são ambíguos nem criptográficos, eles são legíveis e assimiláveis até mesmo pelos programadores iniciantes da equipe – então, se os estagiários entenderem, é a perfeição! :)

Escrever códigos legíveis, claros e óbvios não é uma tarefa árdua – nem uma missão impossível. É só o **hábito** de seguir as melhores práticas – e evitar as piores.

Boas práticas

Boas práticas são todas as atividades desempenhadas por programadores que os levam a escrever códigos de alto nível de qualidade.

Códigos com alto nível de qualidade

Estudar e medir a qualidade de um programa é um desafio para a Engenharia de Software. A qualidade do programa como um todo está ligada aos atributos qualitativos (como facilidade de manutenção, confiabilidade, performance, etc) e métricas (como o tempo necessário para corrigir um *bug*). Na prática, códigos com melhores níveis de qualidade são mais fáceis de manter, demorando menos tempo para, por exemplo, corrigir um *bug*.

As seções a seguir enumeram os tópicos que te ajudarão a programar melhor. São práticas simples, mas eficientes, de organização e escrita do código, que vão desde dar bons nomes até selecionar a melhor assinatura para métodos/funções.

A importância da aderência aos estilos e convenções de código

Todas as linguagens de programação tem convenções e estilos de escrita de código, sejam estas adotadas empiricamente pelos programadores ou as oficialmente documentadas pelo fornecedor da plataforma. As convenções determinam onde se colocam as chaves, quantos espaços são usados para o deslocamento (indentação), como nomear classes, métodos, variáveis e constantes, etc.

A linguagem Java, por exemplo, usa chaves na mesma linha, nomes de métodos no padrão *camelCase* (ou *lowerCamelCase* mais especificamente) e expõe os atributos dos seus objetos segundo o padrão JavaBean[12], com prefixos `get` (ou `is` se for *booleano*) e `set` para métodos. Esses detalhes estão especificados pelo principal mantenedor (Oracle) no documento Java Code Conventions[13]. A Google, enquanto contribuidora, provedora e consumidora de tecnologia Java (ver Android) também oferece um documento chamado Google Java Style[14]. O exemplo a seguir mostra a formatação de um código Java segundo essas convenções:

[12] http://www.oracle.com/technetwork/java/javase/overview/spec-136004.html
[13] http://www.oracle.com/technetwork/java/codeconventions-150003.pdf
[14] https://google-styleguide.googlecode.com/svn/trunk/javaguide.html

Estilos e Convenções para Código Java

```java
// interfaces terminam com "avel"
class LivroTecnico extends Livro implements Publicavel {

    // constantes em CAIXA_ALTA_COM_UNDERLINES
    private static final String IMAGEM_PADRAO = "../img/sem_foto.png";

    // variáveis no formato palavraPalavraPalavra (camelCase)
    private String titulo;
    private int numeroEdicao;

    // propriedades com getPropriedade e setPropriedade
    public String getTitulo() {
        return titulo;
    }

    public void setTitulo(String titulo) {
        this.titulo = titulo;
    }

    public boolean isSemTitulo() {
        return this.titulo == null || titulo.length() == 0;
    }
// ...
```

O código anterior destaca algumas características da linguagem Java, como usar chaves no fim de cada declaração, *get's* e *set's* para expor propriedades, padrão de caixa (minúsculas/maiúsculas) para nomear classes, interfaces, métodos, variáveis e constantes.

Como outro exemplo, considere o código a seguir escrito na linguagem C#. Ele poderia ser escrito exatamente igual ao exemplo anterior, no formato Java, entretanto ele foi escrito segundo o Microsoft C# Guidelines for Names[15] e C# Coding Conventions[16].

Estilos e Convenções para Código C#

```csharp
// interfaces iniciam com "I"
class LivroTecnico : Livro, IPublicavel
{
    private static readonly string IMAGEM_PADRAO = "../img/sem_foto.png";
    private string titulo;
    private int numeroEdicao;

    public string Titulo {
        get {
            return this.titulo;
```

[15] http://msdn.microsoft.com/en-us/library/vstudio/ms229002%28v=vs.100%29.aspx
[16] http://msdn.microsoft.com/en-us/library/ff926074.aspx

```
        }
        set {
            this.titulo = value;
        }
    }

    public int NumeroEdicao {
        get {
            return this.numeroEdicao;
        }
        set {
            this.numeroEdicao = value;
        }
    }

    public bool IsSemTitulo {
        get {
            return this.titulo == null || titulo.Length == 0;
        }
    }
}
```

Como apresentado no código anterior, em C# os métodos iniciam com letras maiúsculas, seguindo padrão *PascalCase* (ou *UpperCamelCase*). Chaves seguem o K&R Style[17], sendo colocadas abaixo em classes e métodos - mas no fim da linha em propriedades e construtos (if's, while's, etc).

É extremamente importante seguir o padrão da linguagem, para não parecer *forasteiro* (o mesmo princípio usado para identificar turistas). Não é incomum programadores migrarem de linguagem e seguirem escrevendo código como se estivesse na linguagem antiga. Por exemplo, o código a seguir mostra um código na linguagem de programação Ruby, primeiro escrito parecido com Java, depois como um programador Ruby nativo faz:

Programando Java em Ruby e Ruby em Ruby

```
# ERRADO! FORASTEIRO JAVA ESCREVENDO RUBY DETECTADO
class Documento

  def setTitulo(titulo)
    @titulo = titulo
  end

  def getTitulo()
    return @titulo
  end

  def setNumeroEdicao(numeroEdicao)
```

[17] http://www.cas.mcmaster.ca/~carette/SE3M04/2004/slides/CCodingStyle.html

```ruby
    @numeroEdicao = numeroEdicao
  end

  def isSemTitulo()
    return @titulo == nil || @titulo.length == 0
  end
# ...

# CERTO!
class Documento

  def titulo=(titulo)
    @titulo = titulo
  end

  def titulo
    @titulo
  end

  def numero_edicao=(numero_edicao)
    @numero_edicao = numero_edicao
  end

  def sem_titulo?
    not defined? @titulo or @titulo.nil? or @titulo.empty?
  end
end
```

As duas classes do exemplo anterior em Ruby funcionam bem! Entretanto, a segunda faz uso dos recursos disponíveis na linguagem Ruby, como `return` opcional, propriedades no formato `propriedade=`, parênteses opcionais em métodos, etc. Ruby usa o estilo determinado pela comunidade de desenvolvimento – e livros técnicos –, mas existem especificações não oficiais como o Ruby Style Guide[18].

Cuidado para não ser identificado como *forasteiro*

Programadores amadores são identificados pelo código que escrevem, por exemplo, destacam-se (negativamente) códigos que não parecem seguir o padrão da linguagem usada.

Como dito no início, toda linguagem tem convenções e guias de estilo, sejam definidas pelos criadores/mantenedores ou voluntariamente pela comunidade de desenvolvedores, como o *PEP-8 Style Guide for Python Code*, que tu podes ver aqui: http://www.python.org/dev/peps/pep-0008/.

[18] https://github.com/rubensmabueno/ruby-style-guide/blob/master/README-PT-BR.md

Programadores Profissionais conhecem e seguem as convenções de código para a linguagem em que estão programando.

Isso é importante, programando em Python escreva código Python, programando em PHP escreva código PHP, e assim por diante. Não seja um Gaijin!

Convenções e guias de estilo adotados pela empresa.

Algumas empresas não seguem a risca as convenções e estilos da linguagem. Suas equipes podem definir seu próprio documento de formato, incluindo novas convenções ou adaptando as existentes – isso funciona bem desde que todos os envolvidos usem a mesma convenção. Algumas, obtêm tanto sucesso na sua especificação, que liberam o documento para a comunidade, caso do Airbnb JavaScript Style Guide[19].

Inventar uma convenção de código particular é uma prática amadora.

Os profissionais, no máximo, decidem com os colegas de equipe, usando argumentação de alto nível, discussões técnicas sofisticadas e respeitando a opinião dos outros – em outras palavras, não abra chaves embaixo dos construtos enquanto todos teus colegas abrem no fim da linha e vice-e-versa.

Busque na Web o *Code Conventions* ou *Code Style Guide* de sua linguagem.

Existem, às vezes, mais de um. Por exemplo, para PHP há o PHP-FIG[20], PEAR[21] e Zend[22], o que importa é aderir a um estilo bem definido.

4 espaços de deslocamento

A linguagem Java, PHP, C# – e outras inspiradas no C/C++ – normalmente usam deslocamento (ou *tab*) de 4 espaços. O deslocamento recomendado é documentado no guia de estilos e convenções. Contudo, nos exemplo que virão a seguir nesse capítulo, usarei 2 espaços com o propósito de compactar o código para caber nas páginas, em cópias impressas e nos *e-readers*, combinado?

O nome é importante

Os nomes devem passar as informações de porque existe, o que faz e como é usado. Muitas informações a respeito do código podem ser reveladas dando um bom nome. Para desenrolar essa seção, vamos começar com um problema simples: nomear uma variável (local ou atributo). A decisão mais simples, menos inteligível e, infelizmente, a primeira a ser selecionada é usar *uma letra*:

[19] https://github.com/airbnb/javascript
[20] http://www.php-fig.org/psr/psr-2/
[21] http://pear.php.net/manual/pt_BR/standards.php
[22] http://framework.zend.com/manual/1.12/en/coding-standard.coding-style.html

```
int d;
```

Não é fácil achar significado para variáveis de uma letra, como `d`, especialmente quando estão fora de um contexto – com exceção das *conhecidas* como `i` em `for(int i = 0;...`, `j`, etc. No exemplo anterior, sabe-se que `d` guarda um número inteiro. Vamos evoluir este exemplo: digamos que o valor de `d` seja 12, então a próxima pergunta é *12 o quê?* Considere que a intenção é demonstrar um tempo decorrido, por exemplo *"12 dias"*. Como a intenção não está clara, uma solução amadora para clarificar é adicionar um comentário:

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

Código comentado, ao contrário do que se pensa, pode ser um mau código.

Quando o código é inteligível o suficiente, os comentários não são necessários. Em outras palavras, os comentários são usados, muitas vezes, para suprir uma carência de inteligibilidade do código.

Poupe o teu tempo em adicionar comentários e, em vez disso, use um nome que revele a intenção. Usar nomes mais adequados poupam a introdução de comentários desnecessários, por exemplo:

```
int tempoDecorridoEmDias;
```

Ah não professor, daí vou ter que escrever mais ...

Foi o que me disse um aluno certa vez quando pedi para trocar o nome curto por um nome expressivo. A variável era algo como `double vp;` que significava *valor a pagar*. Pedi para deixar mais claro, ele trocou para `double vp; // valor a pagar`. Pedi para nomear a variável e ele disse *"Bah, daí vou ter que escrever mais"*. Foi então que eu disse: – Tchê, na verdade tu estás escrevendo menos, já comparaste o *length* da variável comentada `double vp; // valor a pagar` com `double valorAPagar;` – rimos muito, ele ganhou um zero `o_O`, brincadeirinha :)

Estes nomes podem ser ainda mais específicos se houver um significado especial no domínio (e geralmente há), isto é, um sentido particular para as *regras* que estamos implementando. Por exemplo, considere que o tempo decorrido seja contado a partir da última alteração, então um nome adequado para o domínio poderia ser `diasDesdeUltimaAlteracao`. A seguir uma sequência de possíveis nomes:

Ranking de possíveis nomes para uma variável

```
int d; // péssimo, nenhum significado
int tempoDecorrido; // razoável, ambíguo
int tempoDecorridoEmDias; // bom, esclarece "o quê"
int diasDesdeUltimaAlteracao; // ótimo, demonstra o "porquê"
```

Sobre o código anterior, há uma sequência de informações que são adicionadas ao nome da variável. A primeira não diz nada. A segunda diz o quê é armazenado (tempo decorrido). A terceira diz o quê e com qual métrica (dias). A quarta expressa o significado/contexto: o tempo decorrido em dias desde a última alteração.

Opa! Cuidado para não ficar neurótico!

Aprender a dar bons nomes envolve renomear, renomear e renomear, fazendo experimentos. Se não te controlares, isso pode se tornar uma obsessão. Uma característica dos profissionais é que eles sabem parar quando está bom suficiente – coisa que se aprender só com o tempo, não há uma métrica exata.

Como outro exemplo, pense no que faz o código a seguir?

Um método pequeno e obscuro

```
public List<Conta> getCP() {
  List<Conta> c1 = new ArrayList<Conta>();
  for (Conta c2 : cs) if (c2.getTipo() == 2) c1.add(c2);
  return c1;
}
```

Tudo é mais fácil quando os nomes fazem mais sentido. A seguir a mesma funcionalidade escrita com nomes melhores:

O mesmo método anterior iluminado

```
public List<Conta> getContasAPagar() {
  List<Conta> contasAPagar = new ArrayList<Conta>();
  for (Conta conta : todasContas) {
    if (conta.getTipo() == PAGAR) {
      contasAPagar.add(conta);
    }
  }
  return contasAPagar;
}
```

Programadores Profissionais escolhem os nomes judiciosamente.

Programadores Profissionais agem como as polícias de elite: planejam e executam tudo friamente – até a escolha dos nomes.

Está ruim? Refatore.

Alterações feitas no código para torná-lo inteligível chamam-se **refatorações**. As refatorações não alteram a funcionalidade, mas ajudam a tornar o código mais expressivo.

Escrever mais se paga no futuro.

Escrever código inteligível, às vezes, implica em digitar mais. Contudo este esforço é mínimo e se paga nas próximas revisões de código – tenha fé.

> ***"Railãnder, vai já pra casa guri..."***
>
> Era o nome de um colega nos tempos de escola. Hã? O que tem a ver? Dê bons nomes! – e esta regra se estende aos seus filhos.

Diferenças devem ser bem visíveis e as semelhanças agrupadas criteriosamente

Evite usar nomes muito semelhantes para executar atividades bem diferentes. Por exemplo, considere um método `getHistorico` e outro `getHistoricos` em uma classe `Aluno` no código a seguir:

Nomes com diferenças discretas dificultam a percepção de seus objetivos

```
class Aluno {
  List<Historico> getHistorico() { /* ... */ }
  List<Historico> getHistoricos() { /* ... */ }
// ...
```

Qual a diferença entre `getHistorico` e `getHistoricos`? Se o primeiro retorna o histórico do ano atual e o segundo retorna de todos anos, talvez seja melhor tornar as diferenças visíveis, por exemplo:

Nomes distintos facilitam o entendimento e revelam a intenção

```
class Aluno {
  List<Historico> getUltimoHistorico() { /* ... */ }
  List<Historico> getTodosHistoricos() { /* ... */ }
// ...
```

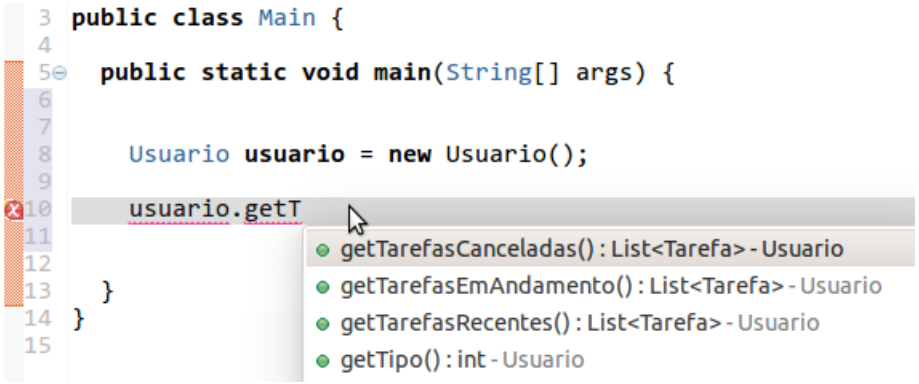
Às vezes podemos projetar alguma semelhança para facilitar a localização dos métodos. Semelhança inicial ajuda a agrupar métodos com propósitos semelhantes, como no exemplo a seguir:

Nomes que agrupam semelhanças e deixam claras as diferenças

```
class Aluno {
  List<Historico> getHistoricoUltimoAno() { /* ... */ }
  List<Historico> getHistoricoTodosAnos() { /* ... */ }
// ...

class Usuario {
  List<Tarefa> getTarefasRecentes() { /* ... */ }
  List<Tarefa> getTarefasEmAndamento() { /* ... */ }
  List<Tarefa> getTarefasCanceladas() { /* ... */ }
// ...
```

Nomes semelhantes também podem facilitar a localização e visualização da API usando ambientes integrados com recursos de autocompletar tais como: Visual Studio, NetBeans, Eclipse, IntelliJ, etc. A ilustração a seguir apresenta o exemplo anterior em uso no Eclipse:



Autocompletar em ação na IDE Eclipse

Na ilustração anterior o editor mostrou todos os métodos que iniciam com `getT`. É o benefício de agrupar, com nomes semelhantes, métodos dentro do mesmo contexto de funcionalidades.

Programadores Profissionais classificam e contextualizam os métodos.

Bem além de pacotes, *namespaces*, classes e outros recursos, **uma prática eficiente de contextualização é agrupar por nomes**. Métodos de um mesmo grupo de operações podem compartilhar semelhanças no nome – se existem muitos métodos semelhantes, eles podem ser extraídos para uma nova classe.

Um outro problema, relacionado a escolher semelhanças ou diferenças, está na sobrecarga de métodos. A sobrecarga deve ser realizada muito criteriosamente. Em alguns casos é melhor escolher nomes diferentes em vez de sobrecarregar. Considere o exemplo:

Mesmo nome com sobrecarga do método exclui

```
1 class AlunoDAO {
2   void exclui(int idAluno) {
3     // código para excluir um aluno com um id específico
4   }
5   void exclui() {
6     // código para excluir todos os alunos da base
7   }
8 // ...
```

A situação apresentada no exemplo mostra um perigo da sobrecarga. Neste exemplo, chamar `alunoDAO.exclui()` (linha 5) implica em eliminar todo os dados da base. Óbvio que ninguém executaria tal método usando o *banco de produção*[23], mas considere uma linha perdida ou esquecida onde o *id* não é passado e que é implantado no cliente – resultado: desastre.

Considere dar assinaturas especiais aos métodos que fazem coisas importantes que não podem ser desfeitas – como excluir tudo! No exemplo a seguir, na linha 5, está a pequena, mas notável, diferença, o nome:

Métodos sem sobrecarga para funcionalidades delicadas

```
1 class AlunoDAO {
2   void exclui(int idAluno) {
3     // código para excluir um aluno com um id específico
4   }
5   void excluiTodos() {
6     // código para excluir todos os alunos
7   }
8 // ...
```

A sobrecarga pode causar o sombreamento de métodos *perigosos*

Não é uma boa prática misturar métodos *"inofensivos"* com métodos *"perigosos"* sem deixar as diferenças bem visíveis.

Programadores Profissionais projetam para que os procedimentos *"delicados"* não sejam executados por acidente.

Este tipo de atividade é semelhante àquela exercida pelos Projetistas de Interfaces Homem-Máquina. Eles realizam decisões de onde colocar e como se comportarão as interfaces – por exemplo, nunca colocar o botão de ligar as luzes e ejetar o assento lado-a-lado nos aviões :)

[23] *Banco de Produção* é a expressão usada para dizer que está sendo usado o Banco de Dados original, real, oficial, etc, com todos os dados do cliente. Normalmente os programadores usam um *Banco de Desenvolvimento* que conta com dados de teste – e evita desastres.

Os nomes segundo os Padrões de Arquitetura e Projeto

Existem padrões de nomes para classes, interfaces, enumerados, etc, de acordo com Padrões de Arquitetura e Projeto de *Softwares* Orientados a Objetos. O nome `AlunoDAO` usado no exemplo é para definir um *Data Access Object* (Objeto de Acesso a Dados) relacionado a uma entidade Alunos – DAO é um padrão de arquitetura. Esse livro não vai discorrer sobre esses nomes, o tema ficará para uma obra futura sobre Princípios e Padrões de Projeto.

Os nomes devem ser pronunciáveis

Tente evitar o uso de siglas ou acrônimos nos nomes. Nomes legíveis facilitam a leitura e a comunicação, ou tu achas fácil dizer: – *Tchê, a variável `prptd` está negativa!* (perdão pela idiossincrasia sulista :).

Nomes pronunciáveis são melhores e permitem a comunicação entre os desenvolvedores. É mais comunicável dizer: – *Tchê, a variável pontoRelativoDePartida está negativa!*. A seguir alguns exemplos com sequências de maus e bons nomes levando em conta a pronúncia:

Sequência de maus e bons nomes segundo a pronúncia

```
double vr; // péssimo, seria Vale Refeição? Realidade Virtual? Volta Redonda?
double valorDeReferencia; // melhor

String dataNasc; // razoável, como se fala? "datanasce" ou "datanasqui"?
String dataNascimento; // melhor

int iu; // ruim, ainda pior se fosse unique identification: "ui"!
int identificacaoUnica; // melhor

ContratoUnico cu; // indecente, impronunciável, mas soletrável
ContratoUnico contratoUnico; // educado, polido
```

Nomes longos e a digitação

O argumento para não escrever nomes muito longos é que eles te fazem digitar mais quando precisa referenciar uma variável, método ou classe. Contudo, usando editores de código modernos, o autocompletar resolve esse dilema.

O caso Bruno Rocha

Em certa empresa que trabalhei, nossos nomes de usuário eram registrados com a primeira letra do nome e o último sobrenome, eu era o *mtorres*, e o Bruno não gostava muito do *username* dele.

Substantivos como nomes de classes e verbos como nomes de métodos

Cada linguagem tem uma convenção de nomes e formato do código (ver Estilos e Convenções de Código), mas no paradigma de Programação Orientada a Objetos *a regra é clara Galvão*: classes são nomeadas como substantivos e métodos como verbos, como no simples exemplo a seguir:

Classes são Substantivos (coisas), Métodos são Verbos (ações)

```
class Aviao {
  void decola() { /* ... */ }
  void pousa() { /* ... */ }
// ...
```

As interfaces podem seguir o mesmo padrão de nomes de classes, ou seja, substantivos, como a `interface Endereco`, ou então seguir um padrão específico da linguagem. Na linguagem Microsoft C#.NET se usa o prefixo `i`. Na linguagem Java se usa adjetivo e o sufixo *"ável"*. Ver exemplos a seguir:

Nomeando interfaces

```
// declarando uma interface
interface Populacao {
  void popular();
}
// com padrão de nomes Microsoft C#.NET:
// "IUmNome"
interface IPopulacao {
  void popular();
}
// com padrão de nomes Java:
// "UmNomeável"
interface Populavel {
  void popular();
}
```

Existem argumentos a favor e contra as técnicas apresentadas no exemplo.

O que importa é ser consistente, ou seja, se as interfaces terão o prefixo `"I"`, então todas as interfaces devem ter o prefixo `"I"`, ok?

Em linguagens Orientadas a Objetos os métodos representam ações que são escritas como verbos no infinitivo ou imperativo, como em `arquivo.salvar()` ou `arquivo.salva()` respectivamente. Usando infinitivo os métodos ficam como no exemplo a seguir:

Métodos nomeados como verbos no infinitivo

```
class Mensagem {
  void enviar() { /* ... */ }
}

class Documento {
  void salvar() { /* ... */ }
  void imprimir() { /* ... */ }
}

// situação de uso:
Documento documento = new Documento();
// imprimir o documento
documento.imprimir();
```

A segunda opção é escrever os métodos usando verbos no imperativo, como no exemplo a seguir:

Métodos nomeados como verbos no imperativo

```
class Mensagem {
  void envia() { /* ... */ }
}

class Documento {
  void salva() { /* ... */ }
  void imprime() { /* ... */ }
}

// situação de uso:
Documento documento = new Documento();
// imprime o documento
documento.imprime();
```

Em inglês não há esse dilema, é sempre no imperativo.

Métodos escritos em inglês são sempre no imperativo como `file.save()` ou `document.print()`. Tu não verás um `file.toSave()` ou `document.toPrint()`. Considere escrever os códigos em português também no imperativo, pense como se fosse uma ordem ao objeto, codificando *"pula personagem"* e *"envia e-mail"* como `personagem.pula()` e `email.envia()`. As próprias linguagens usam o paradigma de programação imperativa[24], onde descrevemos "comandos" passados ao computador.

[24] https://pt.wikipedia.org/wiki/Programa%C3%A7%C3%A3o_imperativa

Independente da escolha, Programadores Profissionais seguem um padrão!

Se `arquivo.salvar()` então `arquivo.fechar()`, não misture os modos. Mesmo que seja Futuro do Presente do Indicativo, se `arquivo.salvarás()` então `arquivo.fecharás()` (hehehe, mas que bobagem :).

Outra convenção comum é usar `verboSujeito` como em `mensagem.enviaMensagem()` e `documento.imprimeDocume`
Nestes casos é redundante e desnecessário, pois `mensagem.envia()` já demonstra a intenção de *enviar a mensagem*. Já em outras situações pode ser bem útil, como em: `janela.mostraTitulo()` ou `janela.removeBorda()`, pois somente `janela.mostra()` ou `janela.remove()` poderia ser mal entendido. A escolha de `verbo` ou `verboSujeito` pode evitar confusões, veja no exemplo a seguir:

Métodos verboSujeito em ação

```
class Cavaleiro extends Personagem {
  void ataca() { /* ... */ } // o Cavaleiro ataca, ok.
  void anda() { /* ... */ }  // o Cavaleiro anda, ok.

  // void guarda() { /* ... */ } // o Cavaleiro guarda, o quê?
  // assim é melhor:
  void guardaEspada() { /* ... */ }
// ...
```

Por omissão, o sujeito do verbo é o objeto.

Um método chamado assim `usuario.valida()` é lido como *"valida usuário"*, já um método chamado assim `usuario.validaSenha()` é lido como *"valida a senha do usuário"*.

Inglês, Português ou русский язык

Este é um assunto bem polêmico. Há quem se sinta confortável em escrever código em inglês e há quem resiste[25]. Existem prós e contras a considerar.

Considere escrever em inglês quando existem nomes de métodos e classes que, devido a convenções e padrões de código e projeto, são melhores escritos e conhecidos em inglês. Por exemplo, os equivalentes do SQL: `insert`, `update`, `delete`, `select`, ou seus pares usados nas classes: `save`, `update`, `remove` e `find`. Palavras típicas para projetar coleções são mais fáceis de introduzir em classes que representem um *container*, como: `size`, `add`, `push` e `pop`, parecem melhor que `tamanho`, `adiciona`, `empurra` e `estoura`. Considere escrever sempre em inglês quando programar um *framework* ou componente de *software* que pode ser usado em vários projetos, hospedado em repositórios públicos (github, gitlab, SourceForge, etc) e usado pela comunidade a nível mundial.

Considere escrever em português quando representa a lógica de negócio para um usuário final. A parte do domínio (*ou negócio*) pode ser difícil escrever em inglês, por não representar

[25] http://youtu.be/rtEaR1JU-ps

bem o *"dialeto"* usado na análise do sistema. Normalmente as classes e métodos usados para modelar o domínio são desenhadas em diagramas, compartilhados com os clientes que, embora não saibam programar, conhecem as entidades envolvidas no seu sistema – mesmo que seja em papel. Nomes em inglês podem dificultar o entendimento dos demais membros do projeto, os *não-programadores* do grupo de *stakeholders*[26]. É importante que entendas que os objetos de domínio devem representar objetos reais daquilo que se está modelando e os nomes devem ser os mais próximos dos objetos reais. Isto é importante para que equipe de desenvolvimento e cliente falem a mesma língua, ou seja, se o cliente tem uma imobiliária e, conforme ele, trabalha com repasse de locação, é melhor que tenhamos uma classe `RepasseLocacao` e não `LeaseForward` – ou um nome em inglês mau colocado como `LocationTransfer` (*transferência do local?!?*).

Programadores Profissionais sabem quando devem escrever em português ou inglês.

A regra simples é usar nomes em português para objetos de domínio e em inglês para extensões da linguagem e utilitários de programação.

Programando para o Mundo

Nomes em inglês são ótimos para criar utilitários, bibliotecas e *frameworks*. Quando o código está disponível abertamente (*open-source*) e seu projeto está disponível para *download* e contribuições, considere códigos em inglês que permitem a colaboração de qualquer pessoa no mundo, já que o inglês é a língua franca[27] dos programadores e profissionais de tecnologia.

Classes e métodos pequenos

Como programadores, nós solucionamos problemas e encapsulamos o código para a solução destes problemas em classes e métodos (ou funções, só para lembrar). Existem boas práticas para implementar classes e métodos, mas uma regra essencial pode ser resumida em uma palavra: **PE-QUE-NO**.

Não existem métricas exatas, mesmo com o sincero esforço do pessoal da Engenharia de Software – mas algumas são bem precisas. Dentro do grupo das imprecisas, minha sugestão é que os métodos devem caber na tela e as classes deveriam ser visualizadas com *meia dúzia de page down* ou *page up*.

Esqueça os instrumentos, *use a força Luke!*

É difícil estabelecer uma regra geral, então quando *"sentir"* que está difícil de acompanhar métodos e classes por causa do comprimento é por que está na hora de *"quebrar"* em partes menores.

[26] Projetos de Software não são desenvolvidos apenas por codificadores, também são considerados membros: clientes, parceiros de negócio, analistas, gerentes de projeto, analistas de TI e outras pessoas que colaboram ou têm interesse de um jeito ou outro. Toda essa *"gente"* é chamada de **Stakeholders**.

[27] Idioma comum usado na comunicação entre indivíduos que, embora tenham diferentes línguas nativas, compartilham dos mesmos interesses http://pt.wikipedia.org/wiki/Lingua_franca

Como um exemplo, considere a implementação de um método para finalizar um pedido na internet. O cliente tem um carrinho, endereço, forma da pagamento e deseja fazer o pedido. É uma mecânica relativamente simples, que pode ser codificada com um método `PedidoService#fazPedido(Carrinho):Pedido`, como pode ser visto no código a seguir:

Um método longo demais para fazer um Pedido

```
class PedidoService {

  public Pedido fazPedido(Carrinho carrinho) {

    if (carrinho.getItens().size() == 0) {
      throw new IllegalArgumentException("Carrinho Vazio");
    }

    if (carrinho.getFormaPagamento() == null) {
      throw new IllegalArgumentException("Forma Pagamento não selecionada");
    }

    if (carrinho.getFormaPagamento() == CARTAO_CREDITO) {
      Cliente cliente = carrinho.getCliente();
      SegurCredWS sc = new SegurCredWS();
      sc.checaCredito(cliente.getCPF().replace(".", "").replace("-","."));
      if (!sc.getSituacao().equals(Situacao.OK)) {
        throw new NoCreditException("Cliente não tem crédito liberado");
      }
    }

    EstoqueService estoqueService = new EstoqueService();
    for (Item item : carrinho.getItens()) {
      if (estoqueService.getQuantidadeEstoque(
        item.getProduto().getCodigo()) < item.getQuantidade()) {
        throw new SemEstoqueParaItem(item);
      }
    }

    // ... mais 30 linhas de transação

    Pedido pedido = new Pedido();
    pedido.setItens(carrinho.getItens());
    pedido.setCliente(carrinho.getCliente());
    pedido.setFormaPagamento(carrinho.getFormaPagamento());

    // ... mais algumas linhas com configurações do pedido

    PedidoDAO pedidoDAO = new PedidoDAO();
    pedidoDAO.salva(pedido);
```

```
    return pedido;
  }
}
```

Métodos longos têm muitas desvantagens, mas o destaque são essas três:

- **difíceis de ler**: visto que todas as linhas (instruções) não cabem na tela;
- **difíceis de manter**: visto que é necessário percorrer toda a classe ou método para localizar problemas e alterar ou introduzir novas funcionalidades;
- **difíceis de reutilizar**: já que todos os passos, mesmos os reaproveitáveis, estão dentro do mesmo bloco.

Métodos com muitas instruções também exercem responsabilidades demais. O segredo para criar um método melhor é decompor as atividades em partes menores (ou sub-atividades), em outras palavras, separar/fracionar as responsabilidades. Isso será visto no tópico a seguir: Um problema de cada vez.

Um problema de cada vez, pequeno gafanhoto

Uma habilidade essencial dos programadores é saber dividir um *"problemão"* em pequenos *"probleminhas"*. Na verdade, não é exclusividade dos programadores. Ciência, atividades de matemática, gestão, engenharias de qualquer espécie e muitas outras áreas trabalham com o fracionamento do *empreendimento*[28].

Na programação existem vários modos de dividir o algoritmo: em arquivos, pacotes, módulos, classes, métodos, funções, etc. Em Java, por exemplo, uma das decisões rotineiras de um programador é escolher em qual classe e com quais métodos um algoritmo será implementado.

Projetar classes é uma tema para o próximo livro, neste, vamos considerar a divisão em métodos. Métodos podem expressar a resolução direta de um problema, por exemplo, a quantidade de dias entre duas datas. Considere o método `Datas#diasEntre(Date,Date):int` sendo usado:

Método utilitário para contar a quantidade de dias entre Datas

```
System.out.println(
  Datas.diasEntre(
    new Date("12/25/2014"),
    new Date("01/05/2015")
  )
); // imprime 11
```

A assinatura do método revela sua responsabilidade essencial (neste caso, calcular dias entre duas datas). Entretanto, sem conhecer a implementação, não sabemos exatamente quantos problemas são resolvidos internamente. Frequentemente, um método resolve vários pequenos problemas, tudo implementado em um único bloco. Considere o método a seguir:

[28] *Nós podemos resolver qualquer problema introduzindo um nível extra de indireção.* Este é o Teorema Fundamental da Engenharia de Software, termo difundido por um grande programador: [Andrew Koenig](https://en.wikipedia.org/wiki/Andrew_Koenig_(programmer).

Método implementado sem divisões do problema

```
1  package monobloco;
2
3  public class Util {
4
5    public static String cameliza(String str) {
6      char[] strCharArray = str.toCharArray();
7      int nroEspacos = 0;
8      for (int i = 0; i < strCharArray.length; i++) {
9        if (strCharArray[i] == ' ') {
10         nroEspacos++;
11       }
12     }
13     char[] strCamelizada = new char[strCharArray.length - nroEspacos];
14     for (int i = 0, j = 0; i < strCharArray.length; i++) {
15       if (strCharArray[i] != ' ') {
16         if (strCharArray[i] >= 65 && strCharArray[i] <= 90) {
17           strCamelizada[j++] = (char) (strCharArray[i] + 32);
18         } else {
19           strCamelizada[j++] = strCharArray[i];
20         }
21       } else {
22         i++;
23         if (strCharArray[i] >= 97 && strCharArray[i] <= 122) {
24           strCamelizada[j++] = (char) (strCharArray[i] - 32);
25         } else {
26           strCamelizada[j++] = strCharArray[i];
27         }
28       }
29     }
30     return new String(strCamelizada);
31   }
32
33   public static void main(String[] args) {
34     System.out.println(cameliza("um texto de teste"));
35   }
36 }
37 // https://github.com/marciojrtorres/tecnicas-praticas-codificacao/blob/master/um\
38 -problema-de-cada-vez/src/monobloco/Util.java
```

O método anterior recebe uma *string* e a *cameliza*, por exemplo, *"um texto qualquer"* torna-se *"umTextoQualquer"*. Para realizar esta funcionalidade o método `cameliza` executa operações bem definidas como contar a quantidade de espaços (código entre as linhas 7 e 12), converter o caractere em minúsculo (código entre as linhas 16 e 20) e converter o caractere para maiúsculo (código entre as linhas entre 23 e 27). Pensando nesse ponto de vista, o método resolve um

problema: *camelizar* uma *string*, no entanto, a implementação tem subproblemas que poderiam ser resolvidos em outros métodos, como no exemplo a seguir:

Método implementado subdividindo o problema

```
1  package particionado;
2
3  public class Util {
4
5    public static String cameliza(String str) {
6      char[] strCharArray = str.toCharArray();
7      int nroEspacos = conta(' ', str);
8      char[] strCamelizada = new char[strCharArray.length - nroEspacos];
9      for (int i = 0, j = 0; i < strCharArray.length; i++) {
10       if (strCharArray[i] != ' ') {
11         strCamelizada[j++] = minuscula(strCharArray[i]);
12       } else {
13         strCamelizada[j++] = maiuscula(strCharArray[++i]);
14       }
15     }
16     return new String(strCamelizada);
17   }
18
19   public static int conta(char c, String str) {
20     char[] strCharArray = str.toCharArray();
21     int ocorrencias = 0;
22     for (int i = 0; i < strCharArray.length; i++) {
23       if (strCharArray[i] == c) {
24         ocorrencias++;
25       }
26     }
27     return ocorrencias;
28   }
29
30   public static char minuscula(char c) {
31     if (c >= 65 && c <= 90) {
32       return (char) (c + 32);
33     } else {
34       return c;
35     }
36   }
37
38   public static char maiuscula(char c) {
39     if (c >= 97 && c <= 122) {
40       return (char) (c - 32);
41     } else {
42       return c;
43     }
```

```
44    }
45
46    public static void main(String[] args) {
47      System.out.println(cameliza("um texto de teste"));
48    }
49  }
50  // https://github.com/marciojrtorres/tecnicas-praticas-codificacao/blob/master/um\
51  -problema-de-cada-vez/src/particionado/Util.java
```

No código anterior alguns problemas foram resolvidos em novos métodos: `conta(char,String):int`, `minuscula(char):char` e `maiuscula(char):char`. A criação de novos métodos é importante para atribuir uma responsabilidade clara para cada método, em outras palavras, **é importante que cada método faça apenas uma tarefa bem definida**.

Programadores Profissionais subdividem os problemas e os codificam.

Os profissionais sabem que a subdivisão implica em métodos com tarefas bem definidas, mais fáceis de implementar, ler, entender, reusar, corrigir e testar.

Cuidado! Indireção excessiva pode causar pigarro, impotência e perda de performance em seus programas

Estamos adicionando indireção ao programa quando criamos um método que chama outro método, e este chama mais um método e por aí vai. Dividir o problema tende a adicionar indireção, pois implica em delegar parte do problema para outro componente que pode delegar parte do problema dele a outro. O que tu precisas saber é que cada camada de indireção implica em perda de performance. Claro que não é algo extraordinário, são dezenas de ciclos de CPUs modernas com bilhões disponíveis (centenas de bilhões), mas o programador experiente tem que conviver com essas decisões e saber até onte a indireção é útil e está se pagando – processo conhecido como *tradeoff*.

Poucos parâmetros, menos é mais nesse caso

Os métodos devem ter o mínimo necessário de parâmetros para cumprir sua tarefa. De novo, não há uma métrica exata, mas métodos com mais de três parâmetros tendem a ficar confusos para entender e difíceis de manter. Por exemplo, futuras intervenções que exijam alterações na quantidade de parâmetros podem causar a propagação da alteração em vários pontos do programa – toda alteração da assinatura (declaração de funções e métodos) causa alterações em cascata. Para exemplificar, considere o seguinte método:

Método com muitos parâmetros

```
1  package problema;
2
3  import java.util.Collections;
4  import java.util.List;
5
6  public class Imovel {
7
8    public static List<Imovel> busca(Integer nroQuartos,
9        Integer nroVagasCarro, Double vlrMaximoCondominio,
10       Double vlrMaximoAluguel, Boolean comFoto,
11       Boolean apartamento) {
12     // código necessário para buscar os imóveis
13     return Collections.emptyList();
14   }
15
16   public static void main(String[] args) {
17     List<Imovel> imoveis = Imovel.busca(2, 2, 800.0, 500.0, false, true);
18   }
19 }
20 // https://github.com/marciojrtorres/tecnicas-praticas-codificacao/blob/master/po\
21 ucos-parametros/src/problema/Imovel.java
```

Sobre o código anterior, a quantidade excessiva de parâmetros no método `busca` (entre as linhas 8 e 11) aumenta o acoplamento e dificulta o entendimento. Aumenta o acoplamento porque obriga o programador a informar todos os parâmetros na mesma ordem e com os tipos determinados (no caso de Java, C# e outras linguagens *tipadas*). Dificulta o entendimento porque a utilização do método pode ser confusa, como podes ver na linha 17 a chamada do método `busca(2, 2, 800.0, 500.0, false, true)` é obscura e não é possível prever quais imóveis serão localizados – sem olhar para a assinatura do método.

Programadores Profissionais sabem que projetar a assinatura do método, incluindo a quantidade de parâmetros, é de extrema importância para o futuro do programa.

A escolha do nome, tipo de retorno, quantidade e ordem dos parâmetros, tudo isso é muito importante. É muito difícil submeter alterações nessas características sem *quebrar* o programa e precisar fazer alterações em seus dependentes.

Existem soluções variadas para esse problema, dependendendo muito da linguagem de programação usada. Uma solução comum é agrupar os parâmetros em uma classe – refatoração conhecida como *Objeto Parâmetro*. A seguir o código refatorado:

Método com um único parâmetro

```
1  package solucao;
2
3  import java.util.Collections;
4  import java.util.List;
5
6  public class Imovel {
7
8    public static class FiltroImovel {
9      public Integer nroQuartos, nroVagasCarro;
10     public Double vlrMaximoCondominio, vlrMaximoAluguel;
11     public Boolean comFoto, apartamento;
12   }
13
14   public static List<Imovel> busca(FiltroImovel filtro) {
15     // código necessário para buscar os imóveis
16     return Collections.emptyList();
17   }
18
19   public static void main(String[] args) {
20
21     FiltroImovel filtro = new FiltroImovel() {{
22       vlrMaximoAluguel = 500.0;
23       nroQuartos = 2;
24       vlrMaximoCondominio = 800.0;
25       apartamento = true;
26       comFoto = false;
27       nroVagasCarro = 2;
28     }};
29
30     List<Imovel> imoveis = Imovel.busca(filtro);
31   }
32 }
33 // https://github.com/marciojrtorres/tecnicas-praticas-codificacao/blob/master/po\
34 ucos-parametros/src/solucao/Imovel.java
```

Objetos Parâmetro trazem benefícios como a facilidade de alterar. Considere a necessidade de acrescentar a informação `quartoEmpregada:boolean`, que alteraria a assinatura do método anterior, mas no método refatorado faríamos a inclusão do atributo na classe `FiltroImovel`, entre as linhas 8 e 12 – e isso não quebraria as chamadas que já existem. Outro benefício é o desacoplamento da ordem. Anteriormente a ordem importava[29], mas neste caso podemos definir qualquer parâmetro em qualquer ordem, como pode ser visto no código anterior entre as linhas 21 e 28.

[29] Acoplamento *posicional* é um dos vários tipos de acoplamento. Uma lista pode ser vista neste endereço https://en.wikipedia.org/wiki/Connascence_(computer_programming)

Em JavaScript, é ainda mais fácil simplificar a quantidade de parâmetros, basta projetar o método para esperar um objeto escrito na notação JSON[30], como no exemplo a seguir:

Objetos Parâmetro em JavaScript

```
// muitos parâmetros == MAIS ACOPLADO:
function busca(nroQuartos, nroVagasCarro, vlrMaximoCondominio,
               vlrMaximoAluguel, comFoto, apartamento) { // rígido
  // implementação ...
}
// ofuscado
busca(2, 2, 800, 500, false, true);


// poucos parâmetros == MENOS ACOPLADO:
function busca(filtro) { // flexível
  var nroQuartos = filtro.nroQuartos;
  var nroVagasCarro = filtro.nroVagasCarro;
  // implementação ...
}
// explícito
busca({vlrMaximoAluguel: 500, nroQuartos: 2,
       vlrMaximoCondominio: 800, apartamento: true,
       comFoto = false, nroVagasCarro = 2});

// ES6 (JavaScript 2015)
function busca({nroQuartos, nroVagasCarro, vlrMaximoCondominio,
               vlrMaximoAluguel, comFoto, apartamento}) { // melhor
  // implementação ...
}
// convencional
busca({vlrMaximoAluguel: 500, nroQuartos: 2,
       vlrMaximoCondominio: 800, apartamento: true,
       comFoto = false, nroVagasCarro = 2});
```

O segundo método `busca` é bem mais flexível que o primeiro. Em outras linguagens, como Ruby, o mesmo comportamento pode ser obtido usando um `hash` (mesmo que `Map` em Java ou `Dictionary` em C#) como parâmetro, considerada um boa prática para métodos com uma lista longa de parâmetros. Outras opções incluem *arrays*, *varargs*, etc.

Programadores Profissionais projetam para que métodos aceitem poucos parâmetros.

Eles tem consciência dos problemas quando se usa muitos parâmetros. Eles conhecem os recursos disponíveis na linguagem para projetar métodos pequenos e com assinaturas concisas e claras.

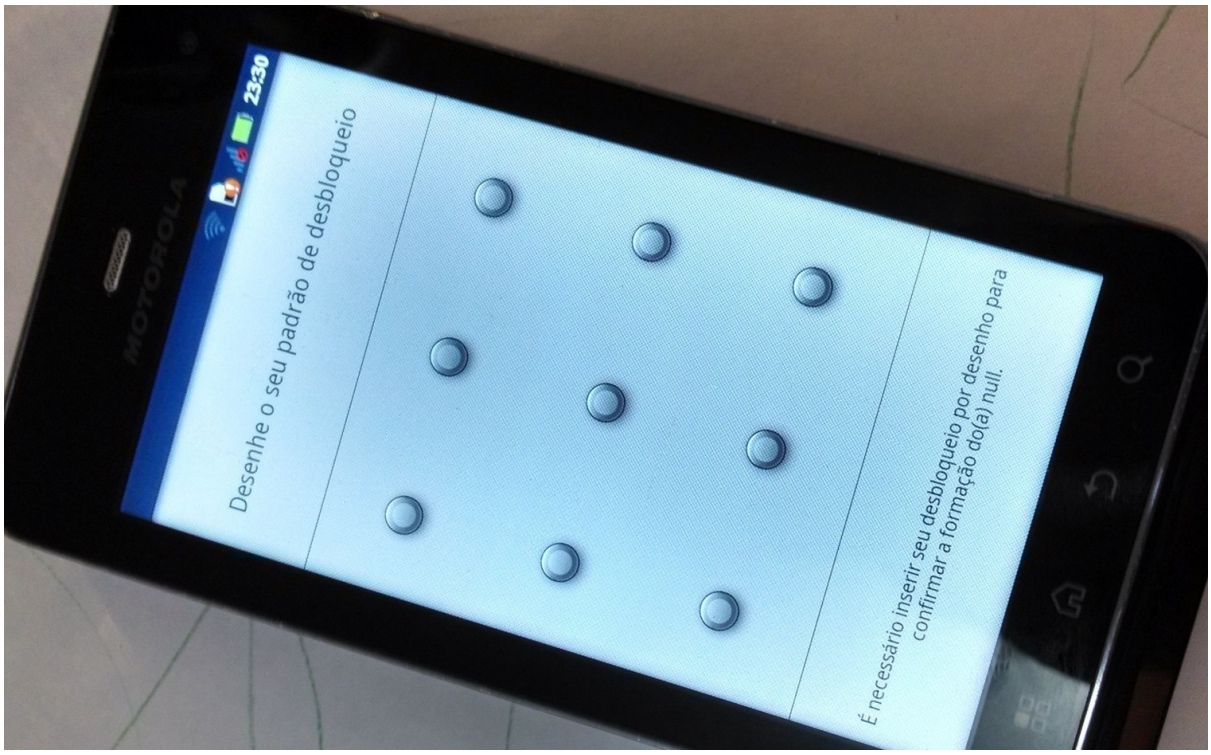
[30] JSON é o acrônimo de JavaScript Object Notation, um construto usado para definir estruturas e instanciar objetos na linguagem JavaScript.

O problema do acoplamento.

O baixo acoplamento é um atributo qualitativo importante nos projetos de software. O objetivo é que cada módulo dependa e conheça o mínimo possível dos demais. A ordem dos parâmetros é um tipo de acoplamento muito perigoso. Por exemplo, se tu alterares o método `busca` para primeiro ler o número de vagas para carros e depois o número de quartos, não *quebraria* a utilização, pois os parâmetros são do mesmo tipo (`int`), mas terias um *bug* muito difícil de localizar. **Frequentemente, é melhor quando o programa quebra** em vez de esconder um *bug*.

Projete para que `NULL` não seja passado como parâmetro

Primeira coisa que eu tenho pra te dizer: `NullPointerException` é a exceção mais enfadonha e, infelizmente, comum. A grande maioria dos programadores deixa passar uma ou outra verificação de nulo. Outros, pior, nem esperam um nulo – e é então que ele aparece (figura a seguir).



Encontre o NULL

O programador tem que lidar com duas realidades. A primeira é aceitar o `null` e lidar com ele. É inevitável. Um `null` *cairá no teu colo* em algum momento (sem pronunciar os eles, essa sentança fica estranha `o_O`). A segunda é que, embora lidemos com ele, não temos que ficar olhando por *nulls* o tempo todo, podemos projetar para que ele seja considerado, mas não obrigatório. Difícil de entender? Vamos a um exemplo: considere um programa que busca produtos entre um valor mínimo e máximo, sendo que o mínimo ou máximo podem ser omitidos – assim só um limite é considerado. A seguir uma implementação que considera o `null` como recurso:

Usando o NULL como recurso

```
1  package problema;
2
3  import java.util.ArrayList;
4  import java.util.List;
5
6  public class Produto {
7
8    private String descricao;
9    private Double valor;
10
11   public Produto(String descricao, Double valor) {
12     this.descricao = descricao;
13     this.valor = valor;
14   }
15
16   public String getDescricao() {
17     return descricao;
18   }
19
20   public void setDescricao(String descricao) {
21     this.descricao = descricao;
22   }
23
24   public Double getValor() {
25     return valor;
26   }
27
28   public void setValor(Double valor) {
29     this.valor = valor;
30   }
31
32   private static List<Produto> produtos =
33       new ArrayList<Produto>();
34
35   static {
36     produtos.add(new Produto("TV LCD", 1500.0));
37     produtos.add(new Produto("TV LCD 3D", 2500.0));
38     produtos.add(new Produto("TV Smart LCD 3D", 2800.0));
39     produtos.add(new Produto("TV Plasma", 1200.0));
40   }
41
42   public static List<Produto> busca(
43       Double vlrMaximo, Double vlrMinimo) {
44
45     if (vlrMinimo == null) {
```

```
46        vlrMinimo = Double.MIN_VALUE;
47      }
48
49      if (vlrMaximo == null) {
50        vlrMaximo = Double.MAX_VALUE;
51      }
52
53      List<Produto> lista = new ArrayList<Produto>();
54
55      for(Produto p : Produto.produtos) {
56        if (p.getValor() != null
57            && p.getValor() >= vlrMinimo
58            && p.getValor() <= vlrMaximo) {
59          lista.add(p);
60        }
61      }
62
63      return lista;
64    }
65
66    public static void main(String[] args) {
67      List<Produto> entre2000e3000 = busca(2000.0, 3000.0);
68      List<Produto> partir2000 = busca(2000.0, null);
69      List<Produto> ateh2000 = busca(null, 2000.0);
70    }
71
72  }
73  // https://github.com/marciojrtorres/tecnicas-praticas-codificacao/blob/master/pr\
74  ojete-para-que-null-nao-seja-passado-como-parametro/src/problema/Produto.java
```

No código anterior, o `null` é esperado como parâmetro (linhas 45 a 51) e utilizado como opção para ignorar o mínimo ou máximo. A busca apenas por mínimo ou máximo pode ser vista nas linhas 68 e 69, ela é feita passando `null` como parâmetro – uma má prática.

Algumas linguagens implementam o Null Safety

Por exemplo, a linguagem Kotlin oferece tipos nuláveis e não-nuláveis. Nela, para uma variável poder ser nula tu deves declarar como nulável adicionando uma `?`, como em: `var nome: String?`.

Os `null`'s literais devem ser evitados por vários motivos. Um deles é a imprevisibilidade. Por exemplo, um método chamado `transfere(null, null, 200.0, null, null)` é menos intuitivo que `transfere("1245-6", "55478-9", 200.0, "1245-6", "77884-3")`, com todos os dados completos.

Ainda, sobre o código no exemplo anterior, o método `busca` pode ser refatorado para não receber nulos como parâmetro. Basta separar os métodos, como pode ser visto no código a seguir:

Desobrigando a passagem de NULL como parâmetro

```
1  package solucao;
2
3  import java.util.ArrayList;
4  import java.util.List;
5
6  public class Produto {
7
8    private String descricao;
9    private Double valor;
10
11   public Produto(String descricao, Double valor) {
12     this.descricao = descricao;
13     this.valor = valor;
14   }
15
16   public String getDescricao() {
17     return descricao;
18   }
19
20   public void setDescricao(String descricao) {
21     this.descricao = descricao;
22   }
23
24   public Double getValor() {
25     return valor;
26   }
27
28   public void setValor(Double valor) {
29     this.valor = valor;
30   }
31
32   private static List<Produto> produtos =
33       new ArrayList<Produto>();
34
35   static {
36     produtos.add(new Produto("TV LCD", 1500.0));
37     produtos.add(new Produto("TV LCD 3D", 2500.0));
38     produtos.add(new Produto("TV Smart LCD 3D", 2800.0));
39     produtos.add(new Produto("TV Plasma", 1200.0));
40   }
41
42   public static List<Produto> buscaValorMinimo(
43       Double vlrMinimo) {
44     return buscaValorEntre(vlrMinimo, null);
45   }
```

```
46
47    public static List<Produto> buscaValorMaximo(
48        Double vlrMaximo) {
49      return buscaValorEntre(null, vlrMaximo);
50    }
51
52    public static List<Produto> buscaValorEntre(
53        Double vlrMinimo, Double vlrMaximo) {
54
55      if (vlrMinimo == null) {
56        vlrMinimo = Double.MIN_VALUE;
57      }
58
59      if (vlrMaximo == null) {
60        vlrMaximo = Double.MAX_VALUE;
61      }
62
63      List<Produto> lista = new ArrayList<Produto>();
64
65      for(Produto p : Produto.produtos) {
66        if (p.getValor() != null
67            && p.getValor() >= vlrMinimo
68            && p.getValor() <= vlrMaximo) {
69          lista.add(p);
70        }
71      }
72
73      return lista;
74    }
75
76    public static void main(String[] args) {
77      List<Produto> entre2000e3000 = buscaValorEntre(2000.0, 3000.0);
78      List<Produto> partir2000 = buscaValorMinimo(2000.0);
79      List<Produto> ateh2000 = buscaValorMaximo(2000.0);
80    }
81
82  }
83  // https://github.com/marciojrtorres/tecnicas-praticas-codificacao/blob/master/pr\
84  ojete-para-que-null-nao-seja-passado-como-parametro/src/solucao/Produto.java
```

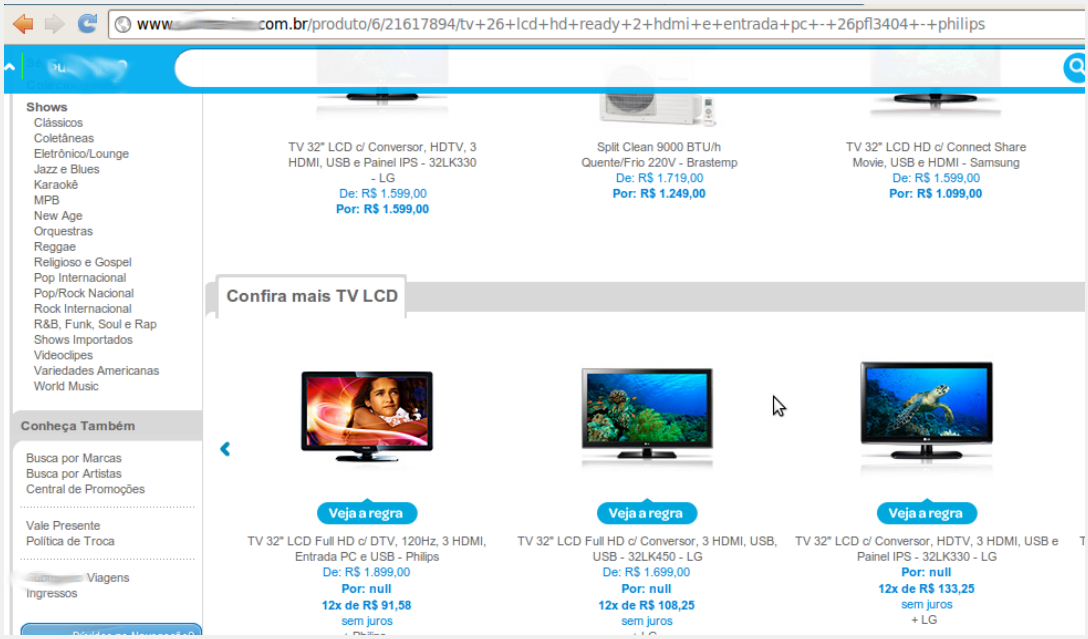
A principal diferença está no código presente entre as linhas 42 e 50. As buscas são mais claras usando métodos dedicados, como pode ser visto entre as linhas 77 e 79. A proposta é que o `null` não seja visto na chamada do método, mas ele pode (deve) ser considerado na implementação (ver linhas 44 e 49).

Programadores Profissionais evitam obrigar a passagem de `null` nos parâmetros.

Eles são conscientes da utilidade e das armadilhas que o `null` oferece, usando-os onde é necessário, mas mantendo-os sob rigoroso controle.

Eu vejo nulos, o tempo todo ...

Daria para escrever um capítulo apenas sobre a gestão do valor nulo. Ele está presente nas linguagens e nos banco de dados e é responsável por *bugs* difíceis de encontrar. Mas a maior inconveniência é quando eles *vazam* para o usuário final.



Vazamento de NULLs :(

Projete para evitar *flags* como parâmetros

É uma boa prática projetar métodos para serem lidos onde são usados. A escolha do nome, quantidade e tipos dos parâmetros, retorno e exceções, todos esses detalhes são importantes para que o código seja expressivo.

Nos métodos *mau projetados* se destacam algumas más práticas, entre elas o uso de parâmetros ou, como é mais conhecido, *argumentos flag*. Eles são parâmetros do tipo booleano usados para realizar uma instrução condicional dentro do método.

Por exemplo, considere um método para contar ocorrências de uma *substring* em uma *string*. A contagem (*count*) pode ser sensível ou não a caixa – ou seja, se serão ou não diferenciadas letras maiúsculas de minúsculas. Usando *argumento flag* uma possível assinatura do método seria `count(String,String,boolean):int` que pode ser visto em uso a seguir:

Quando vale n? O problema dos argumentos flag

```
int n = StringUtil.count("texto",
        "Um texto ou um Grande Texto de exemplo", true);
System.out.println(n); // quanto vale n?
int n2 = StringUtil.count("texto",
         "Um texto ou um Grande Texto de exemplo", false);
System.out.println(n2); // quanto vale n2?
```

As variáveis `n` e `n2` valeriam `1` e `2`, ou `2` e `1`, tudo depende de como o último parâmetro foi projetado e implementado pelo programador, isto é: `true` quer dizer *ignorar a caixa* ou *considerar a caixa*? Métodos com *flags* são abertos a dupla interpretação e não revelam a saída da operação – a não ser que executemos.

A razão para usar *flags* é, normalmente, para economizar código. Considere que a implementação de `StringUtil#count` seja esta a seguir:

Método count usando argumento flag

```
1  package problema;
2
3  public class StringUtil {
4
5    public static int count(String pattern, String text,
6                            boolean isInsensitive) {
7
8      if (isInsensitive) {
9        pattern = pattern.toLowerCase();
10       text = text.toLowerCase();
11     }
12
13     int count = 0;
14     char[] patternChars = pattern.toCharArray();
15     char[] textChars = text.toCharArray();
16
17     for (int i = 0, j = 0; i < textChars.length; i++) {
18       if (textChars[i] == patternChars[j]) {
19         j++;
20         if (j == patternChars.length) {
21           count++;
22           j = 0;
23         }
24       } else {
25         j = 0;
26       }
27     }
28
29     return count;
```

```
30    }
31  }
32  // https://github.com/marciojrtorres/tecnicas-praticas-codificacao/blob/master/ev\
33  ite-argumentos-flag/src/problema/StringUtil.java
```

O uso do *argumento flag* aparece entre as linhas 8 e 11. Na prática, o uso da *flag* adiciona uma obrigação incômoda para quem, na maioria das vezes, quer executar o método considerando a caixa, ou seja, terá de passar sempre `false` como parâmetro, por exemplo como em `count(" ", texto, false)` (contar os espaços). Neste caso não faz diferença se a *flag* é verdadeira ou falsa, mas ela tem de ser informada.

Programadores Profissionais projetam para que parâmetros opcionais sejam de fato opcionais.

Eles codificam para que o comportamento padrão tenha o mínimo de parâmetros introduzindo parâmetros extras para o comportamento configurável em um método sobrecarregado ou outro método similar.

Uma solução recomendada para evitar argumentos *flag* é separar o comportamento em dois métodos, codificando a função da *flag* no nome do método. Neste exemplo, é possível usar `count(String,String):int` e `countIgnoreCase(String,String):int` para diferenciar. Importante, embora existam dois métodos não precisamos de duas implementações! Os dois métodos podem *"terceirizar"* a execução para um método privado auxiliar ou um dos métodos pode transformar a entrada para o segundo, como no exemplo a seguir:

Método count sem argumento flag

```
1   package solucao;
2
3   public class StringUtil {
4
5     public static int countIgnoreCase(String pattern, String text) {
6       return count(pattern.toLowerCase(), text.toLowerCase());
7     }
8
9     public static int count(String pattern, String text) {
10      int count = 0;
11      char[] patternChars = pattern.toCharArray();
12      char[] textChars = text.toCharArray();
13      for (int i = 0, j = 0; i < textChars.length; i++) {
14        if (textChars[i] == patternChars[j]) {
15          j++;
16          if (j == patternChars.length) {
17            count++;
18            j = 0;
19          }
20        } else {
```

```
21          j = 0;
22        }
23      }
24      return count;
25    }
26  }
27  // https://github.com/marciojrtorres/tecnicas-praticas-codificacao/blob/master/ev\
28  ite-argumentos-flag/src/solucao/StringUtil.java
```

O método na linha 5 equivale ao antigo `count` passando `true` como argumento, fazendo o papel do condicional que existia antes.

Em resumo, a ideia geral deste tópico é de que as chamadas a seguir:

sem flags, usando dois métodos

```
int n1 = StringUtil.count("texto",
         "Um texto ou um Grande Texto de exemplo");
int n2 = StringUtil.countIgnoreCase("texto",
         "Um texto ou um Grande Texto de exemplo");
```

São melhores que as que estão a seguir:

com flag, usando um método

```
int n1 = StringUtil.count("texto",
         "Um texto ou um Grande Texto de exemplo", false);
int n2 = StringUtil.count("texto",
         "Um texto ou um Grande Texto de exemplo", true);
```

Programadores Profissionais evitam *flags* em parâmetros de métodos.

Na verdade, os profissionais evitam o uso de *flags* em qualquer lugar. Boa parte das vezes, uma *flag* pode ser substituída por uma solução mais eficiente.

Não introduzirás comentários inúteis

Comentários no código têm sua utilidade e também sua inutilidade. Um código todo *"comentadinho"* nem sempre é um bom sinal, ao contrário, pode ser que os comentários estejam ali para explicar um código que não é claro e expressivo o suficiente.

> ***Bom código é sua própria documentação***
>
> *Bom código é sua própria, e melhor, documentação. Sempre que estiveres por adicionar um comentário, pergunte a si mesmo: "Como eu posso melhorar este código para que esse comentário não seja necessário?"*
>
> – Steve McConnell

Isto talvez mude a tua visão sobre comentar códigos e/ou *pegar* códigos comentados de terceiros. Os tópicos que virão a seguir questionam a necessidade de comentários no código e oferecem alternativas à eles.

Práticas para tornar os comentários obsoletos

A maior parte dos comentários são inseridos no código para explicar o algoritmo ou a lógica do negócio. Alguns são de fato necessários e outros não – se o código fosse suficientemente inteligível. Então, como tornar um comentário desnecessário? Deixando o código inteligível, claro. Considere o exemplo a seguir:

Explicando uma lógica com comentários

```
class Contabil {

  private ReportService rs;

  // método para relatório de lançamentos não contabilizados
  void relatorio() {

    // lançamentos deste ano, desde o início do ano até a data atual
    List<Lancamento> l1 = lancamentoService.lancamentos(
        new Date(new Date().getYear(), 0, 1), new Date());

    // lançamentos não contabilizados
    List<Lancamento> l2 = new ArrayList<Lancamento>();

    for (Lancamento l : l1) {
      // 0: nao contabilizado, 1: contabilizado
      // se está contabilizado
      // 0: ativo, 1:cancelada, 2:inativo
      // tem que estar não contabilizado e não cancelado para enviar
      // para o relatório
      if (l.getStatus() == 0 && l.getArquivo() != 1) l2.add(l);
    }

    rs.send(l2);
  }
}
```

O código do exemplo anterior tem muitos comentários com o propósito de explicar a lógica. Pequenas alterações (refatorações) podem tornar os comentários desnecessários, como renomear o método e variáveis. A seguir uma versão do mesmo código, que ainda requer melhorias, mas sem comentários e com a mesma clareza:

Explicando uma lógica com o próprio código

```
1  class Contabil {
2
3    private ReportService reportService;
4
5    private static final int NAO_CONTABILIZADO = 0;
6    private static final int CANCELADO = 1;
7
8    void relatorioLancamentosNaoContabilizados() {
9
10     Date hoje = new Date();
11     Date inicioAno = new Date(hoje.getYear(), 0, 1);
12
13     List<Lancamento> lancamentosDoAno =
14       lancamentoService.lancamentos(inicioAno, hoje);
15
16     List<Lancamento> lancamentosNaoContabilizados =
17       new ArrayList<Lancamento>();
18
19     for (Lancamento lancamento : lancamentosDoAno) {
20       if (lancamento.getStatus() == NAO_CONTABILIZADO
21           && lancamento.getArquivo() != CANCELADO) {
22         lancamentosNaoContabilizados.add(lancamento);
23       }
24     }
25
26     reportService.send(lancamentosNaoContabilizados);
27   }
28 }
```

O código no exemplo anterior tem a mesma expressividade do código comentado – até mais. É indispensável escolher bons nomes para métodos, parâmetros e variáveis (linhas 6, 11 e 14), introduzir constantes (linhas 3, 4, 18 e 19) para evitar *números mágicos* e adicionar variáveis explicativas (linhas 8 e 9) para clarificar expressões confusas. Em outras palavras, o código foi todo refatorado – técnicas de refatoração serão vistas no Capítulo 11: Melhorando Códigos Existentes;

Programadores Profissionais escrevem códigos que não precisam de comentários.

Todos os profissionais sabem que um código de qualidade é mais do que um algoritmo que funciona. Os profissionais escrevem e re-escrevem códigos para que sejam autoexplicativos.

Não comente o óbvio

Alguns comentários são irrelevantes, ocupam espaço, atrapalham a legibilidade e funcionam como ruído no código, como no exemplo a seguir:

Comentários irrelevantes atrapalham a leitura

```
class Cliente {
  int codigo; // código do cliente
  String nome; // nome do cliente

  // construtor vazio
  public Cliente() {
  }
// ...
```

Não são necessários comentários para informar de quem são os atributos e que o construtor vazio é, bem, vazio. Comentários desse tipo são elegantemente convidados a retirarem-se – não há o que substituir ou refatorar nesses casos.

Outros comentários irrelevantes são aqueles inseridos no código para, ou por, quem está aprendendo, por exemplo:

Comentários que explicam a sintaxe da linguagem são irrelevantes

```
// para cada empréstimo em emprestimos
for (Emprestimo e : emprestimos) {
  report.add(e);
  if (report.hasErrors()) {
    break; // sai do for
  }
}
```

A declaração `for (Elemento e : elementos)` do Java já quer dizer *para cada elemento em elementos*. Programadores Profissionais, confortáveis com a linguagem, não introduzem estes comentários – como o: `break` sai do laço `for` (isso é até uma ofensa! Lembre que em grandes empresas o código é compartilhado entre os programadores, ou seja, **o código não é teu** e deves seguir o padrão de codificação da equipe, isto é, tu tens a responsabilidade de manter o nível.

Programadores Profissionais não comentam a sintaxe da linguagem.

Eles são proficientes na linguagem que estão trabalhando e quando têm alguma dúvida usam material técnico externo ao código – como folhas de referência.

Comentários amadores e o Marketing Pessoal.

Introduzir comentários amadores no código vai atrapalhar a tua reputação e a escalada nos planos de cargos e salários. Lembre sempre: *tu és o código que tu escreves.*

A saga do aprendiz.

Ser aprendiz é duro, eu sei, todos somos aprendizes, isso nunca termina, sempre há mais, sobretudo na computação. A seguir algumas dicas rápidas e diretas para aprendizes:

- Como iniciante, considere colecionar alguns livros e guias de referência rápida;
- Faça anotações, mas não no código que estás trabalhando e que vai ser implantado;
- Toque projetos pessoais (*toy projects*), e nesses sim, comente a vontade;
- Converse com colegas da área sobre técnicas e práticas;
- Leia (ouça, veja) material técnico especializado, blogs, revistas, podcasts, vídeo tutoriais, etc;
- E **o mais importante**: **ajudem alguém!** (tu irás te surpreender com o quanto se aprende enquanto se ensina).

Use o teu talento de escritor para escrever a *documentação* em vez de simplesmente adicionar *comentários*

Documentar o código é uma prática profissional – e saudável. A maioria das documentações de código são introduzidas como comentários formatados, mas com uma diferença essencial: eles ficam disponíveis nas IDE's e permitem gerar arquivos (HTML, PDF, etc) com a documentação.

A maior parte das linguagens possui um padrão para documentação de classes, atributos e métodos. Por exemplo, na linguagem Java todo o conteúdo de documentação é colocado entre `/**` (*barra asterisco asterisco*) e `*/` (*asterisco barra*), como no código a seguir:

Método documentado

```java
package utils;

/**
 * Classe utilitária com métodos estáticos
 * usados para lidar com strings.
 *
 * @author Márcio Torres
 * @version 1.0
 *
 */
public class StringUtil {

  /**
   * padchar padrão
   */
  public static final char ESPACO = ' ';
```

```
17
18    /**
19     * Centraliza uma string dado um número de
20     * colunas e caractere de preenchimento, por exemplo:
21     *
22     * <code>
23     * center("java", 10, '#').equals("###java###")
24     * </code>
25     *
26     * @param string
27     *        string a ser centralizada
28     * @param columns
29     *        número de colunas
30     * @param padchar
31     *        caractere de preenchimento
32     *
33     * @return
34     * a string centralizada precedida e seguida
35     * pelo caractere de pad até o número de colunas
36     *
37     * @throws IllegalArgumentException
38     * se a largura for menor que o comprimento da string
39     */
40    public static String center(
41        String string, int columns, char padchar) {
```

Como prática de documentação, considere sempre escrever um resumo dos objetivos da classe, identificar-se como autor, definir versão e documentar todos os atributos e métodos públicos.

Documentação vs Codificação

Os textos devem ser explicativos, úteis como um tutorial bem escrito. Programadores enraizados no código, introspectivos e pouco sociáveis têm dificuldade em escrever documentação adequada. Para isso dar certo, é necessário pensar um pouco fora-da-caixa em como uma pessoa veria a documentação. A atividade de documentar pode, inclusive, ser delegada para outro membro da equipe.

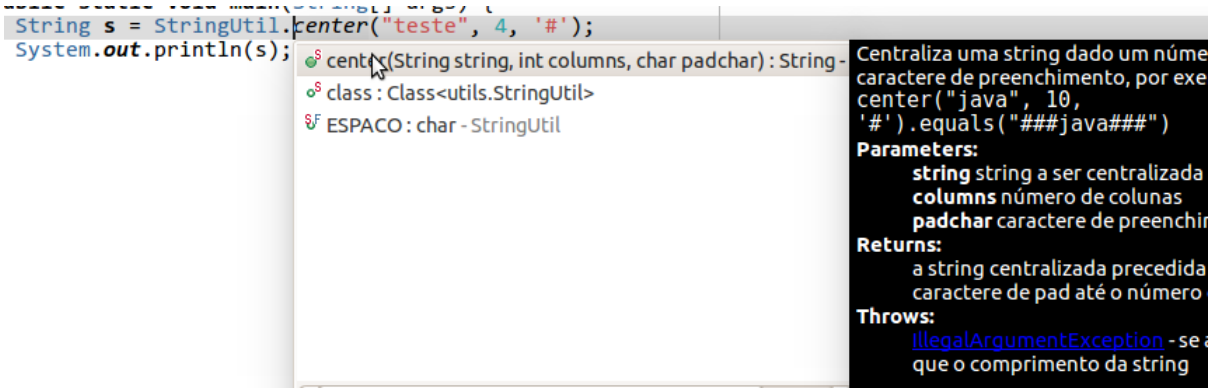
Na linguagem Java são usados alguns metadados, como no código anterior (os itens com `@`), para adicionar algumas informações padronizadas. Por exemplo, para documentar os métodos existem metadados específicos: os parâmetros do método são explicados com o metadado `@param`, o retorno (quando não é `void`) é explicado com o metadado `@return` e as exceções que podem acontecer usando o metadado `@throws` (lança) – ver o código anterior entre as linhas 18 e 39.

A documentação pode ser exportada para um documento externo, com o uso de uma ferramenta. Na plataforma Java é usado o JavaDoc[31] – um utilitário que gera o HTML a partir do código

[31] O JavaDoc é um utilitário presente no Java SDK, mais sobre ele no endereço: http://www.oracle.com/technetwork/java/javase/documentation/index-jsp-135444.html

documentação. A documentação da API da Java[32] foi construída com esse recurso e ferramenta.

Documentar traz algumas vantagens bem interessantes. Uma delas é que o documento exportado pode ser disponibilizado junto com o projeto – no diretório junto com a base de código ou em um servidor web na intranet/internet. Programadores recém contratados podem ler essa documentação para obter informações sobre as classes do projeto – também é útil para retomar o projeto e adicionar funcionalidades ou consertar *bugs*. Outra vantagem é a integração da documentação nas IDE's como NetBeans, Eclipse ou Intellij IDEA, que mostram as informações a respeito das classes e métodos durante o uso, como pode ser visto na figura a seguir:



Documentação sendo exibida na IDE Eclipse

Outras linguagens possuem métodos e ferramentas semelhantes para gerar a documentação a partir do código-fonte, tais como:

- **phpDocumentor**: para a linguagem PHP, disponível em http://www.phpdoc.org/.
- **Sphinx**: para Python, disponível em http://sphinx-doc.org/.
- **rdoc**: para Ruby, disponível em http://rdoc.sourceforge.net/.
- **MS XML Documentation Comments**: disponibilizada junto com a plataforma .NET e usada na linguagem C#, por exemplo. Detalhes sobre ela em http://msdn.microsoft.com/en-us/library/b2s063f7.aspx.
- **Doxygen**: disponível em http://www.doxygen.org/, pode ser usada para várias linguagens entre elas: C, Objective-C, C#, PHP, Java, Python, Fortran, VHDL, etc.

Programadores Profissionais documentam o código, simples assim.

Programadores *calejados* sabem que seus colegas e outros programadores usarão seu código apenas se conseguirem entendê-lo. API's sem documentação ou com documentação deficiente tendem ao fracasso.

Documentação vs Popularidade

Todas as grandes bibliotecas e *frameworks* populares devem grande parte da sua popularidade à sua documentação abrangente. Empresas e até entusiastas de novidades esperam que uma nova ferramenta disponibilizada no mercado ou comunidade seja pelo menos minimamente documentada.

[32] http://docs.oracle.com/javase/6/docs/api/

Considerações sobre boas práticas

Práticas são resultados de experiências que dão certo (e são seguidas) e que dão errado (e então são evitadas). Com bastante exercício, as práticas viram hábitos e tornam-se uma cultura. Cultivar e compartilhar práticas, princípios e padrões com sua equipe de desenvolvimento é o primeiro passo para se tornar um líder técnico – um profissional que toda empresa deseja e precisa.

Todo programador sério se preocupa em escrever códigos de alta qualidade. Este capítulo trouxe algumas práticas mais comuns, frequentes e essenciais para quem precisa programar para viver. Sendo sincero, quando concebi este capítulo ele tinha mais práticas, entretanto eu queria escrever um livro pequeno que não fosse uma literatura vasta e exaustiva do tema.

Existem outros livros que se dedicam a enumerar e discorrer sobre boas práticas de programação. Entre eles posso citar três excelentes: *Código Limpo* de Robert Martin, *Code Complete* de Steve McConnel e *A Arte de Escrever Programas Legíveis* de Dustin Boswell e Trevor Foucher.

No próximo capítulo *a saga continua* com algumas técnicas, sofisticadas ou pouco triviais, usadas por Programadores Profissionais para escrever códigos expressivos, seguros e rápidos.

Capítulo 010 – Técnicas de Codificação

Bons hábitos...

Eu não sou um ótimo programdor; Eu sou só um bom programador com ótimos hábitos.

– Kent Beck

Após programar por bastante tempo tu percebes que alguns entraves nos códigos se repetem com bastante frequência, seja na implementação de lógicas de domínio, interfaces com o usuário, persistência de informações e outras partes. Esses entraves são tão comuns que todos os programadores passam por eles uma hora ou outra, acabam descobrindo um padrão e, frequentemente, chegam a mesma solução, uma *solução-padrão*. De fato, essas soluções comuns são conhecidas como padrões de codificação/implementação e podem ser aplicadas na maioria das linguagens de programação.

Neste capítulo estão presentes algumas *soluções-padrão* descritas como Técnicas de Codificação. Elas servem para, por exemplo, melhorar a inteligibilidade, maximizar a performance, minimizar a repetição de código e algumas para, de fato, resolver o problema mais profissionalmente – ou elegantemente.

Foram separadas as técnicas mais populares e úteis, descrevendo-as junto com o problema, a motivação para usá-las e exemplos de como implementá-las[33]. Espero que aproveites bem esse capítulo e faças bom uso no teu dia-a-dia.

Encadeamento de construtores (*constructor chaining*)

A sobrecarga de construtores é um recurso comum nas linguagens modernas e permite instanciar um objeto com diferentes parâmetros de inicialização. O problema é que, com mais de um construtor, a lógica pode ficar espalhada, requerendo medidas para controlar a repetição de código.

A técnica de *encadeamento de construtores* permite que um objeto seja instanciado com quantidade variável de parâmetros, caracterizando opcionalidade e valores padrão (*defaults*) e, o mais importante, permite **sobrecarregar construtores sem incorrer na repetição de código**.

Como exemplo, considere uma classe `TempoDecorrido` e suas instâncias que podem ser construídas/inicializadas com 3 argumentos (horas, minutos e segundos) ou 2 (minutos e segundos) ou mesmo 1 argumento (só os segundos). A duplicação de código está em cada atividade de construção, para validar se não é negativo e também para adequar/converter minutos e segundos acumulados. Veja a listagem a seguir:

[33] Como já foi dito, a maioria dos exemplos são escritos na linguagem Java, mas são aplicáveis a praticamente todas as linguagens de programação – com um pequeno ajuste e exceção aqui e ali.

Três construtores com código duplicado entre eles

```
package problema;

public class TempoDecorrido {

  private int horas, minutos, segundos;

  public TempoDecorrido(int horas, int minutos, int segundos) {
    if (horas < 0) {
      throw new IllegalArgumentException("horas negativas " + horas);
    }
    if (minutos < 0) {
      throw new IllegalArgumentException("minutos negativos " + minutos);
    }
    if (segundos < 0) {
      throw new IllegalArgumentException("segundos negativos " + segundos);
    }
    if (segundos >= 60) {
      minutos += segundos / 60;
      this.segundos = segundos % 60;
    } else {
      this.segundos = segundos;
    }
    if (minutos >= 60) {
      horas += minutos / 60;
      this.minutos = minutos % 60;
    } else {
      this.minutos = minutos;
    }
    this.horas = horas;
  }

  public TempoDecorrido(int minutos, int segundos) {
    if (minutos < 0) {
      throw new IllegalArgumentException("minutos negativos " + minutos);
    }
    if (segundos < 0) {
      throw new IllegalArgumentException("segundos negativos " + segundos);
    }
    if (segundos >= 60) {
      minutos += segundos / 60;
      this.segundos = segundos % 60;
    } else {
      this.segundos = segundos;
    }
    if (minutos >= 60) {
```

```
      this.horas = minutos / 60;
      this.minutos = minutos % 60;
    } else {
      this.minutos = minutos;
    }
  }

  public TempoDecorrido(int segundos) {
    if (segundos < 0) {
      throw new IllegalArgumentException("segundos negativos " + segundos);
    }
    if (segundos >= 60) {
      this.minutos = segundos / 60;
      this.segundos = segundos % 60;
    } else {
      this.segundos = segundos;
    }
    if (this.minutos >= 60) {
      this.horas = this.minutos / 60;
      this.minutos = this.minutos % 60;
    }
  }
```

O código anterior mostra a vantagem da sobrecarga de construtores, permitindo construir/inicializar o `TempoDecorrido` informando horas, minutos e segundos com `new TempoDecorrido(3,40,10)`, horas e minutos com `new TempoDecorrido(40,10)` e apenas segundos com `new TempoDecorrido(10)`. Contudo, eles usam as mesmas validações e regras e têm a desvantagem de possuir códigos duplicados que poderiam ser compartilhados.

A mecânica da solução começa por estabelecer um construtor completo, com todos os argumentos necessários para uma inicialização completa e válida do objeto. Então, implementa-se novos construtores com menos argumentos, que *repassam* os argumentos recebidos e completam as *lacunas* com *defaults*. Em Java, a chamada `this(param_1, param_2, param_n)` é usada para acorrentar (*enchain*) um construtor ao outro. O resultado pode ser visto no código a seguir:

Construtores encadeados

```
package solucao;

public class TempoDecorrido {

  private int horas, minutos, segundos;

  public TempoDecorrido(int horas, int minutos, int segundos) {
    if (horas < 0) {
      throw new IllegalArgumentException("horas negativas " + horas);
    }
    if (minutos < 0) {
```

```
      throw new IllegalArgumentException("minutos negativos " + minutos);
    }
    if (segundos < 0) {
      throw new IllegalArgumentException("segundos negativos " + segundos);
    }
    if (segundos >= 60) {
      minutos += segundos / 60;
      this.segundos = segundos % 60;
    } else {
      this.segundos = segundos;
    }
    if (minutos >= 60) {
      horas += minutos / 60;
      this.minutos = minutos % 60;
    } else {
      this.minutos = minutos;
    }
    this.horas = horas;
  }

  public TempoDecorrido(int minutos, int segundos) {
    this(0, minutos, segundos);
  }

  public TempoDecorrido(int segundos) {
    this(0, segundos);
  }
```

As vantagens ficam claras quando se compara os dois códigos de exemplo: o encadeamento reaproveita a lógica principal, reduz o tamanho e a duplicação de código.

Programadores Profissionais reaproveitam a lógica e evitam duplicar código.

Fazer o encadeamento de construtores é uma técnica usada para alcançar esses objetivos.

Existem outros meios para implementar opcionalidade de parâmetros na inicialização.

A opcionalidade de parâmetros também pode ser obtida por valores *default* em vez de sobrecarga de construtor. Em PHP, por exemplo, não há sobrecarga, mas há valor *default* para parâmetro – há um exemplo em encadeamento-de-construtores/src/php/TempoDecorrido.class.php[34]

[34] https://github.com/marciojrtorres/tecnicas-praticas-codificacao/blob/master/encadeamento-de-construtores/src/php/TempoDecorrido.class.php

Não se engane, CTRL + C e CTRL + V são seus piores inimigos!

A maior parte dos entraves relacionados ao projeto do *software* aparecem durante as correções e adições de funcionalidades. O código duplicado é uma das fontes mais frequentes de problemas, como alterações em cascata e *bugs* difíceis de encontrar. A causa mais comum do código duplicado é a apologia e cultura do CTRL + C e CTRL + V, cuidado!

Interface fluente (*fluent interface*)

A interface de um objeto é representada pelos seus métodos, que são usados para enviar comandos e fazer consultas aos objetos. Os métodos definem operações e, às vezes, os objetos precisam de várias operações em sequência para completar uma atividade. Esse sequenciamento de operações pode se beneficiar da técnica de **Interface Fluente**. A técnica consiste em encadear uma chamada de método na outra, reduzindo a quantidade de código necessário para executar operações sequenciais sobre uma instância.

A Interface Fluente é muito popular nos *frameworks* e bibliotecas modernas como, por exemplo, o JodaTime[^joda-time], que usa uma Interface Fluente para a configuração de seus objetos, como pode ser visto no exemplo a seguir:

Interface fluente do framework JodaTime

```
1 DateTimeFormatter dateTimeFormatter = new DateTimeFormatterBuilder()
2     .appendDayOfMonth(2).appendLiteral('-')
3     .appendMonthOfYearShortText()
4     .appendLiteral('-').appendTwoDigitYear(1956)
5     .toFormatter();
```

O exemplo anterior mostra o encadeamento de métodos entre as linhas 2 e 5. Os métodos pertencem a interface do objeto *builder* (um Padrão de Projeto popular) para `DateTimeFormatter` (uma classe do JodaTime para formatar datas). Para notar melhor como o uso de interfaces fluentes simplifica a configuração em sequência, observe o código usando a abordagem tradicional a seguir :

Sem interface fluente é necessário declarar o objeto todas as vezes

```
DateTimeFormatterBuilder builder = new DateTimeFormatterBuilder();
builder.appendDayOfMonth(2);
builder.appendLiteral('-');
builder.appendMonthOfYearShortText();
builder.appendLiteral('-');
builder.appendTwoDigitYear(1956);
DateTimeFormatter dateTimeFormatter = builder.toFormatter();
```

O *"segredo"* de `DateTimeFormatterBuilder` é que ele retorna a instância em todos os métodos `append`. A técnica de interface fluente também é usada em algumas classes da API do Java, como a `StringBuilder` que pode ser vista em ação a seguir:

StringBuilder quando usada sem e com interface fluente

```
    // NÃO FLUENTE:
    StringBuilder strBuilder = new StringBuilder();
    strBuilder.append("Data: ");
    strBuilder.append("___/___/___");
    strBuilder.append(" ");
    strBuilder.append("Nome: ");
    strBuilder.append("__________________");
    String str = strBuilder.toString();
    System.out.println(str);

    // FLUENTE:
    StringBuilder strBuilder2 = new StringBuilder();
    String str2 = strBuilder2
        .append("Data: ")
        .append("___/___/___")
        .append(" ")
        .append("Nome: ")
        .append("__________________")
        .toString();
    System.out.println(str2);
  }
```

A mecânica da implementação de interfaces fluentes não é tão complicada quanto parece. É possível até mesmo implementar em classes que já existem refatorando os métodos de configuração (*setters*, *appenders*, etc) declarando o tipo do próprio objeto como retorno (os comandos geralmente são `void`) e adicionando um `return this;` (o próprio objeto) no fim do método. Como exemplo, considere a seguinte classe e sua utilização:

Uma classe normal, com getters e setters

```
7  public class HtmlForm {
8
9    private String id;
10   private String action;
11   private String method;
12   private List<String> classes = new ArrayList<String>();
13   private List<HtmlElement> elementos = new ArrayList<HtmlElement>();
14
15   public void setId(String id) {
16     this.id = id;
17   }
18
19   public void setAction(String action) {
20     this.action = action;
21   }
```

```
22
23    public void setMethod(String method) {
24      this.method = method;
25    }
26
27    public void addClass(String clazz) {
28      this.classes.add(clazz);
29    }
```

Usando o exemplo anterior, preste atenção aos *setters*, como o `setId(String):void` na linha 19. Para usar `HtmlForm` é preciso construir o objeto e *chamar* vários métodos para configurá-lo, como pode ser visto na classe `Main` a seguir:

Usando a classe HtmlForm e HtmlInput para construir um formulário HTML

```
package normal;

public class Main {
  public static void main(String[] args) {

    HtmlInput input1 = new HtmlInput();
    input1.setId("usuario");
    input1.setName("usuario");
    input1.setType("text");

    HtmlInput input2 = new HtmlInput();
    input2.setId("senha");
    input2.setName("senha");
    input2.setType("password");

    HtmlInput input3 = new HtmlInput();
    input3.setValue("Login");
    input3.setType("submit");

    HtmlForm form = new HtmlForm();
    form.setAction("/login.php");
    form.setMethod("post");
    form.addClass("fancy");
    form.addClass("shadowed");
    form.setId("form-login");
    form.addElement(input1);
    form.addElement(input2);
    form.addElement(input3);

    String f = form.getHtmlString();

    System.out.println(f);
```

```
  }
}
```

As classes `HtmlInput` e `HtmlForm` precisam de configuração em sequência. O objetivo é refatorá-las para usar a técnica de Interface Fluente. A refatoração introduz pequenas mudanças como, por exemplo, `HtmlForm#setId(String):void` em `HtmlForm` para `HtmlForm#setId(String):HtmlForm` e a adição de um `return this;` no fim do método. Sendo mais específico é preciso fazer o que está no código a seguir:

```
// alterar o método setId de:
public class HtmlForm {
  // ...
  public void setId(String id) {
    this.id = id;
  }
  // ...
// para:
public class HtmlForm {
  // ...
  public HtmlForm setId(String id) {
    this.id = id;
    return this;
  }
  // ...
```

Compreendendo a mecânica da alteração, é só aplicar o mesmo à todos os métodos de configuração. O resultado da classe `HtmlForm` totalmente refatorada para ser *fluente* está no excerto de código a seguir:

Classe HtmlForm com métodos fluentes

```
public class HtmlForm {

  private String id;
  private String action;
  private String method;
  private List<String> classes = new ArrayList<String>();
  private List<HtmlElement> elementos = new ArrayList<HtmlElement>();


  public HtmlForm setId(String id) {
    this.id = id;
    return this;
  }

  public HtmlForm setAction(String action) {
    this.action = action;
```

```
    return this;
  }

  public HtmlForm setMethod(String method) {
    this.method = method;
    return this;
  }

  public HtmlForm addClass(String clazz) {
    this.classes.add(clazz);
    return this;
  }

  public HtmlForm addElement(HtmlElement element) {
    this.elementos.add(element);
    return this;
  }
```

Após estas mudanças, construir e configurar um `HtmlForm` com vários campos *input* torna-se bem mais simples, resultado da chamada de métodos com a Interface Fluente, como pode ser visto no código a seguir:

Usando a Interface Fluente para construir e configurar um HtmlForm

```
package fluente;

public class Main {
  public static void main(String[] args) {

    String form = new HtmlForm()
        .setAction("/login.php")
        .setMethod("post")
        .addClass("fancy")
        .addClass("shadowed")
        .setId("form-login")
        .addElement(
            new HtmlInput().setId("usuario")
              .setName("usuario").setType("text"))
        .addElement(
            new HtmlInput().setId("senha")
              .setName("senha").setType("password"))
        .addElement(new HtmlInput()
          .setValue("Login").setType("submit"))
        .getHtmlString();

    System.out.println(form);
```

```
  }
}
```

Programadores Profissionais consideram usar Interfaces Fluentes em objetos que precisam de muitos parâmetros de configuração.

Eles sabem que interfaces fluentes diminuem bastante a quantidade de código e eliminam redundâncias.

A técnica de Interface Fluente pode ser combinada com a técnica Método Fábrica Estático.

Algumas técnicas podem ser combinadas para produzir melhores resultados. Por exemplo, Interface Fluente pode ser usada junto com Métodos Fábrica Estáticos para substituir construtores (técnica que será vista mais a frente neste capítulo), por exemplo, tranformando este trecho `new HtmlInput().setId("usuario").setName("usuario").setType("text")` em `HtmlInput.name("usuario").setId("usuario").setType("text")`.

Funções variádicas (*varargs*)

Escolher a quantidade (e tipos) de parâmetros é uma decisão importante ao projetar métodos (ou funções). Para reduzir o acoplamento e minimizar pontos de mudança, uma diretriz é usar a menor quantidade possível de parâmetros. Entretanto, alguns métodos podem ser projetados para aceitar desde nenhum até uma quantidade ilimitada de parâmetros (método/função variádica), importante, sem incorrer nos problemas de acoplamento.

Revisitando o problema da assinatura do método

A quantidade de parâmetros já foi discutida no tópico Poucos Parâmetros no Capítulo 1.

O objetivo da técnica de funções variádicas é não impor limites no método, evitando ter de reprojetá-lo no futuro. Em outras palavras, *se um método pode aceitar 3 parâmetros e depois ser reprojetado para 4 e mais tarde 5, então porque não aceitar então um número ilimitado de uma vez?*

Princípios de Projeto

Existem alguns Princípios de Projeto que ajudam a fazer boas decisões antes e durante a escrita do código. Algumas técnicas cumprem alguns princípios. Por exemplo, as funções variádicas ajudam a escrever métodos/funções concordantes com o Princípio de Projeto *Zero One Infinity*.

Zero One Infinity Rule (ZOI)

A *Regra do Zero Um Infinito* (tradução livre) é um princípio de projeto de *software* proposto por Willem Louis van der Poel (imponente nome, não?) argumentando que não deve ser imposto qualquer limite arbitrário de instâncias de qualquer entidade quando faz mais sentido deixar ilimitado. O nome é resultado da diretriz: não permitir instâncias (*zero*) ou permitir uma (*one*) e, se precisar de mais, então que se permita qualquer número de instâncias (*infinity*). Em outras palavras, o objetivo é não assumirmos limites arbitrários quando, de fato, o limite não é conhecido ou necessário. Por exemplo, podemos imaginar a modelagem de bancos de dados seguindo esse princípio, como inicialmente definir que um *contato* tem um *telefone* e depois reprojetar para aceitar dois *telefones*; neste caso, não faz sentido limitar a dois telefones, pois se pode dois então pode três; se pode três então pode quatro; então pode *n* (onde almoçam dois, almoçam três, quatro, ..., é só botar mais água no feijão!).

Como funções (ou métodos) variádicas aceitam de zero a vários parâmetros, elas são uma forma simples e elegante de implementar opcionalidade (em vez de restrição) de parâmetros. No caso de métodos, é possível permitir um número variável de parâmetros sem precisar fazer sobrecarga.

Como exemplo, considere uma classe e método utilitário que permite gerar um *caminho de pão*[35], usado principalmente em aplicações Web. Como chamar a classe de `MigalhaDePao` pareceria estranho (ver Inglês, Português ou русский язык) preferi o nome conhecido *Breadcrumb*. O objetivo do método é gerar a seguinte saída:

```
Nível#0 separador Nível#1 separador Nível#2 (...)
Ex.:
Home
Home > Livros
Home > Livros > Informática
```

Considere que a especificação inicial solicita um caminho de até 3 níveis (0, 1 e 2), então o objetivo é projetar classe e métodos que suportem nenhum, um ou dois parâmetros para gerar o caminho. Primeiro, observe uma implementação rápida usando a sobrecarga de métodos no código a seguir:

Implementando parâmetros variáveis com sobrecarga

```
package sobrecarregado;

public class Breadcrumb {

  private String level0;
  private String separator;

  public Breadcrumb(String nivel0, String separator) {
    this.level0 = nivel0;
    this.separator = separator;
```

[35] Caminho de Pão é o nome popular usado para definir a navegação estrutural. Em inglês (e pela comunidade de *design*) é conhecido como *breadcrumb* (migalhas de pão; o nome vem do conto Hänsel_und_Gretel).

```
  }

  public Breadcrumb() { this("Home", " > "); }

  public String get() { return level0; }

  public String get(String level1) {
    return new StringBuilder(level0)
      .append(separator).append(level1)
      .toString();
  }

  public String get(String level1, String level2) {
    return new StringBuilder(get(level1))
      .append(separator).append(level2)
      .toString();
  }

  public static void main(String[] args) {

    Breadcrumb caminho = new Breadcrumb();
    // Home
    System.out.println(caminho.get());
    // Home > Livros
    System.out.println(caminho.get("Livros"));
    // Home > Livros > Informática
    System.out.println(caminho.get("Livros", "Informática"));

  }
}
```

Importante, a solução com sobrecarga funciona, mas não é o ideal (lembre: não estamos usando a métrica simplista *funciona ou não*). É uma solução útil e eficiente para o momento, mas não é "**a prova de futuro**". Pensando como um programador que projeta, não é adequado codificar um limite arbitrário em algo que é aparentemente ilimitado (mesmo que a equipe de análise diga que o limite é 3, não acredite neles :P). Revisitando o exemplo do Caminho de Pão, novos níveis podem ser necessários no futuro, por exemplo, considere a alteração necessária para introduzir mais um nível no exemplo a seguir:

Implementando mais um parâmetro adicionando outra sobrecarga

```
public String get(String level1, String level2, String level3) {
  return new StringBuilder(get(level1, level2))
    .append(separator).append(level3)
    .toString();
}
```

Agora, pense na introdução de 10 níveis, teríamos que escrever mais 6 métodos! A essa altura acho que já percebeste que é inviável (ou pelo menos enfadonho) fazer a sobrecarga, então uma solução mais profissional é não impor limites, projetando o método para ser *"aberto"* (flexível) a uma quantidade *"infinita"* (teoricamente) de parâmetros.

Programadores Profissionais projetam para flexibilidade.

Eles sabem que a imposição de limites arbitrários adiciona rigidez aos programas. Os profissionais fazem projeções, tentando antever mudanças e limites não são amigáveis a mudanças.

A mecânica para implementação de um método variático é solicitar primeiro os parâmetros obrigatórios e por fim declarar o último como variádico. Os problemas deste tópico, resolvido antes com sobrecarga, estão resolvidos no exemplo a seguir usando um único método variádico:

Implementando número variável de argumentos com métodos variádicos

```
1  package variadico;
2
3  public class Breadcrumb {
4
5    private String level0;
6    private String separator;
7
8    public Breadcrumb(String nivel0, String separator) {
9      this.level0 = nivel0;
10     this.separator = separator;
11   }
12
13   public Breadcrumb() { this("Home", " > "); }
14
15   public String get() { return level0; }
16
17   public String get(String... levels) {
18     StringBuilder builder = new StringBuilder(level0);
19     for (String level : levels) {
20       builder.append(separator)
21              .append(level);
22     }
```

```
23      return builder.toString();
24    }
25
26
27    public static void main(String[] args) {
28      Breadcrumb caminho = new Breadcrumb();
29      // Home
30      System.out.println(caminho.get());
31      // Home > Livros
32      System.out.println(caminho.get("Livros"));
33      // Home > Livros > Informática
34      System.out.println(caminho.get("Livros", "Informática"));
35      // Home > Livros > Informática > Banco de Dados
36      System.out.println(caminho.get("Livros", "Informática", "Banco de Dados"));
37      // Home > Livros > Informática > Banco de Dados > PostgreSQL
38      System.out.println(caminho.get("Livros", "Informática", "Banco de Dados", "Po\
39  stgreSQL"));
40    }
41  }
```

Em Java, os argumentos variáveis são chamados de *varargs* e implementados como `Tipo... param` (linha 17). O parâmetro é recebido como um *array*, então o argumento variável deve ser percorrido para ser lido (linhas 19 à 22). Muitas linguagens suportam argumentos variáveis, por exemplo, em Python é usado um `*` no argumento variádico como pode ser visto no código a seguir:

Funções Variádicas com varargs em Python

```
class Breadcrumb(object):
  def __init__(self, level0 = "Home", separator = " > "):
    self.level0 = level0
    self.separator = separator
  def get(self, *levels):
    if len(levels) == 0:
      return self.level0
    path = [self.level0]
    path.extend(levels)
    return self.separator.join(path)

caminho = Breadcrumb()
# Home
print caminho.get()
# Home > Livros
print caminho.get("Livros")
# Home > Livros > Informatica
print caminho.get("Livros", "Informatica")
# Home > Livros > Informatica > Banco de Dados
```

```
print caminho.get("Livros", "Informatica", "Banco de Dados")
# Home > Livros > Informatica > Banco de Dados > PostgreSQL
print caminho.get("Livros", "Informatica", "Banco de Dados", "PostgreSQL")
```

Funções variádicas e as linguagens de programação

Algumas linguagens (por exemplo: Python, Ruby e PHP) não suportam sobrecarga de funções/métodos, mas o mesmo objetivo pode ser atingido com o uso habitual de *varargs*. Em outras linguagens, mesmo com suporte, não se observa a *cultura* (o hábito dos programadores) de usar *varargs* (C e C++ por exemplo) e é mais comum ver a sobrecarga (ou ponteiros e *struct's*).

Programadores Profissionais questionam a análise/especificação.

Hoje o mundo do trabalho precisa mais do generalista do que do especialista. Foi-se o tempo em que havia a segregação da equipe entre Analistas e Programadores, desde quando se observou que o *programador* tinha deficiências por não ter habilidades de análise e que o *analista* tinha deficiências por não ter habilidades de programação (uma das motivações para os cursos de Análise e Desenvolvimento). O trabalho do Programador Profissional não é *só codificar*, ele pode (deve) ter conhecimentos e habilidades de análise e projeto e **deve ser capaz de questionar a especificação do sistema**. Isto é, observar requisitos ambíguos, especificações incompletas e quaisquer outras informações que possam ser mal entendidas. Bugs na análise/especificação podem ser identificados no momento da implementação e as funcionalidades não devem ser entregues enquanto não forem totalmente esclarecidas. Um erro na especificação é mais fácil (e mais barato) de corrigir do que um erro implementado, que torna-se ainda muito mais custoso de corrigir depois do sistema ser implantado.

Iteradores e iterabilidade

Todas as linguagens disponibilizam uma ou mais estruturas de dados que *guardam* vários objetos. O tipo mais comum é o *array* (vetor), mas existem outras bem populares como: listas (*List*), conjuntos (*Set*) e mapas ou dicionários (*Map*, *Dictionary*, *Hash*).

Em Java, as estruturas de dados são disponibilizadas através da API *Collections* (coleções), que implementa cada tipo, com algumas opções, por exemplo, para listas há as classes `ArrayList` e `LinkedList`. A variedade de opções para estruturas de dados pode, por vezes, engessar o código em vez de flexibilizar, pois cada uma pode possuir uma maneira particular de operar. Por exemplo, considere o seguinte código:

Diferentes estruturas de dados requerem diferentes estratégias de iteração

```
// ITERANDO UM ARRAY:
String[] linguagens =
  new String[] {"Java", "Ruby", "C#", "PHP", "Python"};
for (int i = 0; i < linguagens.length; i++) {
  System.out.println(linguagens[i]);
}
// ITERANDO UMA LISTA:
List<String> linguagens =
  Arrays.asList("Java", "Ruby", "C#", "PHP", "Python");
for (int i = 0; i < linguagens.size(); i++) {
  System.out.println(linguagens.get(i));
}
```

Visualizando o código anterior percebe-se o problema: os *arrays* são percorridos de uma maneira e as listas de outra. Isso quer dizer que se escolheres usar *array* inicialmente para implementar, digamos, os produtos em um carrinho de compras e depois mudar para `List` ou `Set`, também terá de mudar a forma de iterar (ou não?).

Ou não! A partir do Java 5 (ou 1.5) foi introduzido o *foreach* (para cada) ou o *for melhorado* (tradução livre de *enhanced for*). Na prática, ele se aproveita de uma interface comum chamada `Iterable` implementada por todas as coleções na linguagem Java e permite iterar todas as coleções de um modo uniforme como no código a seguir:

Mesma estratégia de iteração com For Each

```
1  // ITERANDO UM ARRAY:
2  String[] linguagens =
3    new String[] {"Java", "Ruby", "C#", "PHP", "Python"};
4  for (String linguagem : linguagens) {
5    System.out.println(linguagem);
6  }
7  // ITERANDO UMA LISTA:
8  List<String> linguagens =
9    Arrays.asList("Java", "Ruby", "C#", "PHP", "Python");
10 for (String linguagem : linguagens) {
11   System.out.println(linguagem);
12 }
```

A vantagem das estruturas iteráveis (que implementam `Iterable`) pode ser observada no código entre as linhas de 4 à 6 e 10 à 12: o mesmo *for* é usado para percorrer as duas estruturas. Objetos iteráveis permitem fazer a consulta sem saber como são implementados e/ou sem revelar a estrutura interna ou preocupar-se com ela. Além disso, é possível implementar seus próprios objetos iteráveis implementando a interface `Iterable`.

`Iterable` é uma interface usada para implementar o Padrão de Projeto (*Design Pattern*) *Iterator*

Não é objetivo desse livro abordar *Design Patterns*, entretanto é importante saber que `Iterable` é uma interface usada para cumprir o padrão *Iterador*. Os *Design Patterns* são soluções populares, documentadas e reconhecidas, para problemas comuns de Projeto Orientado a Objetos.

A mecânica para tornar um objeto iterável é fazer sua classe implementar a interface `Iterable<T>` e escrever o método `iterator():Iterator<T>`. Normalmente a interface `Iterator<T>` é implementada em uma classe interna privada, contendo a lógica necessária para o *foreach* percorrer os elementos.

Como exemplo, considere a classe `Disciplina` e propriedades como nome e notas do primeiro, segundo, terceiro e quarto bimestres, mais a nota do exame. As notas poderiam ser armazenadas em uma lista, mas para este exemplo elas serão codificadas como atributos de `Disciplina`. A versão inicial da classe (não iterável) pode ser vista a seguir:

Consultando as notas cadastradas para a disciplina no modo tradicional

```
package nao_iteravel;

public class Disciplina {

  private String nome;
  private Double nota1oBim, nota2oBim,
    nota3oBim, nota4oBim, notaExame;

  public Disciplina(String nome) {
    this.nome = nome;
  }

  public String getNome() {
    return nome;
  }

  public Double getNota1oBim() {
    return nota1oBim;
  }

  public void setNota1oBim(Double nota1oBim) {
    this.nota1oBim = nota1oBim;
  }

  public Double getNota2oBim() {
    return nota2oBim;
  }
```

```
  public void setNota2oBim(Double nota2oBim) {
    this.nota2oBim = nota2oBim;
  }

  public Double getNota3oBim() {
    return nota3oBim;
  }

  public void setNota3oBim(Double nota3oBim) {
    this.nota3oBim = nota3oBim;
  }

  public Double getNota4oBim() {
    return nota4oBim;
  }

  public void setNota4oBim(Double nota4oBim) {
    this.nota4oBim = nota4oBim;
  }

  public Double getNotaExame() {
    return notaExame;
  }

  public void setNotaExame(Double notaExame) {
    this.notaExame = notaExame;
  }

  public static void main(String[] args) {
    Disciplina disciplina = new Disciplina("Engenharia de Software");
    disciplina.setNota1oBim(8.2);
    disciplina.setNota2oBim(6.8);
    disciplina.setNota3oBim(7.1);
    // consultando as notas cadastradas
    if (disciplina.getNota1oBim() != null) {
      System.out.println(disciplina.getNota1oBim());
    }
    if (disciplina.getNota2oBim() != null) {
      System.out.println(disciplina.getNota2oBim());
    }
    if (disciplina.getNota3oBim() != null) {
      System.out.println(disciplina.getNota3oBim());
    }
    if (disciplina.getNota4oBim() != null) {
      System.out.println(disciplina.getNota4oBim());
    }
```

```
    if (disciplina.getNotaExame() != null) {
      System.out.println(disciplina.getNotaExame());
    }
  }
}
```

A versão *não iterável* de `Disciplina` precisa invocar o *getter* de cada nota para consultá-las. Fazendo diferente, a proposta é permitir a iteração da disciplina, ou seja, permitir percorrê-la. Considere que percorrer a disciplina é percorrer as notas fazendo *for (Double nota : disciplina)*. A seguir o exemplo da classe `Disciplina` em sua versão *iterável*:

Iterando as notas cadastradas para a disciplina

```
public class Disciplina implements Iterable<Double> {

  private class NotasIterator implements Iterator<Double> {

    private int i = 0;

    @Override
    public boolean hasNext() {
      if (i == 0 && nota1oBim != null) return true;
      if (i == 1 && nota2oBim != null) return true;
      if (i == 2 && nota3oBim != null) return true;
      if (i == 3 && nota4oBim != null) return true;
      if (i == 4 && notaExame != null) return true;

      return false;
    }

    @Override
    public Double next() {
      i = i + 1;
      if (i == 1 && nota1oBim != null) return nota1oBim;
      if (i == 2 && nota2oBim != null) return nota2oBim;
      if (i == 3 && nota3oBim != null) return nota3oBim;
      if (i == 4 && nota4oBim != null) return nota4oBim;
      if (i == 5 && notaExame != null) return notaExame;
      throw new IndexOutOfBoundsException();
    }

    @Override
    public void remove() {
      throw new UnsupportedOperationException();
    }
  }
```

```
  @Override
  public Iterator<Double> iterator() {
    return new NotasIterator();
  }
```

O que possibilita um objeto ser iterável é a codificação dos três requisitos: implementar `Iterable<T>` (linha 5), implementar `Iterator<T>` (linhas de 7 a 36) e instanciá-lo (linhas de 40 a 42). Este código não é o melhor exemplo de eficiência e eficácia, mas demonstra a mecânica de como construir objetos iteráveis. Com esta implementação é possível consultar as notas da disciplina iterando-a como no exemplo a seguir:

Iterando as notas cadastradas para a disciplina

```
1   public static void main(String[] args) {
2     Disciplina disciplina = new Disciplina("Engenharia de Software");
3     disciplina.setNota1oBim(8.2);
4     disciplina.setNota2oBim(6.8);
5     disciplina.setNota3oBim(7.1);
6
7     for (Double nota : disciplina) {
8       System.out.println(nota);
9     }
10
11  }
```

Embora possas codificar teus objetos iteráveis *"no braço"*, na maior parte das vezes tu podes delegar para um *container* do Java que já implementa `Iterable`. Por exemplo, considere uma classe `NotaFiscal` que agregue vários itens (instâncias de `Item`). É possível tornar a classe `NotaFiscal` iterável permitindo e facilitando a consulta dos itens da nota (considere que percorrer a nota signifique consultar seus itens). A seguir o código que implementa esse conceito:

Repassando o iterador

```
package delegar_iteracao;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

public class NotaFiscal implements Iterable<Item> {

  private int numero;
  private String cliente;
  private List<Item> itens = new ArrayList<Item>();

  public NotaFiscal(int numero, String cliente) {
    this.numero = numero;
```

```
    this.cliente = cliente;
  }

  public void addItem(Item item) {
    this.itens.add(item);
  }

  @Override
  public Iterator<Item> iterator() {
    // REPASSANDO O ITERATOR DE ARRAYLIST
    return itens.iterator();
  }

  public static void main(String[] args) {
    NotaFiscal notaFiscal = new NotaFiscal(12345, "Cliente de Exemplo") {{
      addItem(new Item() {{
        codigo = 1; descricao = "TV LED Samsung";
        preco = 1224.49; quantidade = 1.0;
      }});

      addItem(new Item() {{
        codigo = 1; descricao = "BluRay Player Philips";
        preco = 297.99; quantidade = 1.0;
      }});
    }};
    // ITERANDO A NOTA:
    for (Item item : notaFiscal) {
        System.out.println(item.descricao);
    }
  }
}
```

Programadores Profissionais preferem expor os objetos pelas suas interfaces.

O nível de acoplamento (engessamento?) é menor quando os métodos expõem interfaces em vez de classes concretas. Por exemplo, expor o `Iterator` de uma estrutura é melhor que expor a própria estrutura – seja ela *array*, `Properties`, `List`, `Set`, etc.

Iteradores e Geradores andam juntos

Os mesmos construtos usados para definir objetos iteráveis também são usados para definir objetos geráveis, como será visto a seguir em Geradores.

Iterabilidade em outras linguagens

Muitas outras linguagens suportam iterabilidade, porém com outras interfaces. Ruby, por exemplo, suporta a mesma funcionalidade de iterabilidade definindo-se o método `each` e incluindo o módulo `Enumerable` (na verdade o módulo *enumerable* disponibiliza muito mais funcionalidades).

Geradores (*generators*)

Enquanto iteradores permitem percorrer elementos de uma coleção existente, geradores permitem percorrer itens que são computados sob demanda. Geradores são estruturas populares e usadas com frequência em algumas linguagens, como Python e Ruby, que disponibizam uma instrução especial chamada *yield*, usada para externar um elemento computado para o laço que chamou a função/método. A seguir um exemplo de como gerar a sequência de fibonacci[36] sob demanda em Python:

Geradores em Python: fibonacci

```
def fib(max = 2^16-1):
  fib = 0
  res = 1
  while fib < max:
    print 'info: gerando'
    temp = res
    res = res + fib
    fib = temp
    yield fib

for n in fib(100):
  print 'info: percorrendo'
  print n
```

A computação do próximo número não é realizada enquanto a iteração não avança no laço. Essa é a característica essencial de um gerador.

Esse comportamento pode ser implementado em praticamente qualquer linguagem. Na linguagem Java é possível usar a iterabilidade para implementar um gerador. Considere o gerador da sequência de fibonacci escrito em Java a seguir:

<<Geradores em Java: fibonacci[37]

Os geradores parecem mais naturais em Python (que tem outros recursos legais como *list comprehension*) do que em Java. Entretanto é uma técnica que podes usar sempre que precisares acessar recursos sob demanda.

[36] Sequência de Fibonacci é uma sequência de números inteiros onde cada número subsequente corresponde a soma dos dois anteriores (0, 1, 1, 2, 3, 5, 8, ...).

[37] code/geradores/src/Fib.java

Melhor pedir perdão do que permissão

Esta técnica é muito conhecida na comunidade Python, mas pode ser aplicada em qualquer linguagem. Pedir perdão em vez de permissão é uma metáfora usada para representar a execução de uma operação sem antes ver se é possível ou permitido, capturando eventuais exceções (*pedindo perdão*).

It's Easier Ask Forgiveness than Permission (EAFP).

Esta técnica é conhecida e difundida pelo acrônimo EAFP.

O oposto de outra técnica.

EAFP considera o oposto de Olhe antes de pular, que será visto no tópico seguinte.

A implementação é simples, executa-se a instrução sem qualquer verificação prévia, dentro de um bloco `try/catch` para o caso de exceções.

Como exemplo, considere uma classe `Retangulo` que pode ser instanciada pelo construtor `new Retangulo(400, 300)` ou `new Retangulo("400x300")`. O segundo construtor deve converter as *strings* em inteiros. Em Java isso é feito usando o método `Integer.parseInt(String):int`. O complicador é que algumas *strings* de entrada podem ser inválidas. Para esses casos largura e altura devem assumir `0`. É o momento de fazer uma escolha: validar a entrada ou fazer a conversão diretamente? Seguindo a técnica EAFP a conversão é feita sem verificação, como no código a seguir:

EAFP: executando instruções sem verificações prévias

```
1  public class Retangulo {
2    private int largura;
3    private int altura;
4
5    public Retangulo(int largura, int altura) {
6      this.largura = largura;
7      this.altura = altura;
8    }
9
10   public Retangulo(String laString) {
11     String[] la = laString.split("x");
12     try {
13       this.largura = Integer.parseInt(la[0]);
14       this.altura = Integer.parseInt(la[1]);
15     } catch (Exception e) {
16       this.largura = 0;
17       this.altura = 0;
18     }
```

```
19    }
20
21    public int getLargura() {
22      return largura;
23    }
24
25    public int getAltura() {
26      return altura;
27    }
28
29    @Override
30    public String toString() {
31      return largura + "x" + altura;
32    }
33
34    public static void main(String[] args) {
35      System.out.println(new Retangulo(480, 300));
36      System.out.println(new Retangulo("480x300x100"));
37      System.out.println(new Retangulo("480q300"));
38      System.out.println(new Retangulo("480xqwe"));
39    }
40  }
```

Sem usar o bloco `try/catch`, entre as linhas 12 e 18, seria necessário verificar a validade da *string*, por exemplo, se o *array* tem dois elementos após o *split* e se ambos são inteiros, tudo antes de efetuar o `parseInt`.

Esse mesmo exemplo poderia usar Olhe antes de pular. Na prática, usando *"meia dúzia de ifs/elses"* dá para obter o mesmo efeito sem ter de lidar com exceções.

Muitos profissionais consideram essa técnica uma má prática.

O argumento é de que usar blocos `try/catch` para realizar desvios condicionais é uma subversão do construto, ou seja, o argumento é que os blocos `try/catch` estão sendo usados no lugar onde `if/else` seriam mais adequados. Ademais, `try/catch` tem menor performance que `if/else`.

Em Python, por exemplo, EAFP é aplicável quando é necessário acessar uma chave em um dicionário. Em vez de verificar antes se existe a chave, se faz o acesso diretamente, mas abraçando a instrução em um `try/except` (um pedido de perdão?). Considere um contador de palavras em Python como o que está a seguir:

EAFP em Python

```
1  def conta(frase):
2      palavras = frase.split()
3      contagem = {}
4      for word in palavras:
5          try:
6              contagem[word] += 1
7          except KeyError, e:
8              contagem[word] = 1
9      return contagem
10
11 print conta("texto exemplo texto livro")
```

O trecho entre as linhas 5 e 8 caracterizam o EAFP. Ele poderia ser diferente, por exemplo fazendo um `if word in contagem` antes de incrementar, entretanto essa técnica é muito popular entre os *pythonistas*.

Olhe antes de pular

Assim como EAFP, LBYL (*Leap Before You Leap*) é popular na comunidade Python, mas a técnica de *"olhar antes de pular"* é aplicável em praticamente qualquer linguagem de programação.

Muito provavelmente já usaste esta técnica – só não sabias que deram um nome para ela. Sua mecânica é muito simples, consiste em checar ou validar um dado antes executar uma ação sobre ele, normalmente abraçando a instrução com um `if`, sendo o estilo mais tradicional de Programação Defensiva[38].

> ***"se existem 3 maneiras de algo dar errado, todos acontecerão ou só a que der mais prejuízo"***
>
> A Lei de Finagle é conhecida pela declaração: *"Qualquer coisa que possa dar errado, dará"*. É um corolário das Leis de Murphy, mais conhecidas, como *"Se alguma coisa pode dar errado, dará. E mais, dará errado da pior maneira, no pior momento e de modo que cause o maior dano possível."* ou *"As variáveis variam menos que as constantes"*. O ensinamento que tiramos disto é: valide, sempre!

Como LBYL é o contraste de EAFP, veremos os mesmos exemplos usados em *Melhor Pedir Perdão que Permissão*. Considere a classe `Retangulo` usando a técnica LBYL:

[38] Programação defensiva é a adição de contigências com o objetivo de antecipar circunstâncias excepcionais, mantendo a funcionalidade do *software*.

LBYL: verificando se é seguro antes de executar instruções

```java
public class Retangulo {
  private int largura;
  private int altura;

  public Retangulo(int largura, int altura) {
    this.largura = largura;
    this.altura = altura;
  }

  public Retangulo(String laString) {
    String[] la = laString.split("x");
    if (la.length == 2) {
      if (ehInteiro(la[0]) && ehInteiro(la[1])) {
        this.largura = Integer.parseInt(la[0]);
        this.altura = Integer.parseInt(la[1]);
      } else {
        this.largura = 0;
        this.altura = 0;
      }
    }
  }

  private boolean ehInteiro(String s) {
    for (char c : s.toCharArray()) {
      if (!Character.isDigit(c)) {
        return false;
      }
    }
    return true;
  }

  public int getLargura() {
    return largura;
  }

  public int getAltura() {
    return altura;
  }

  @Override
  public String toString() {
    return largura + "x" + altura;
  }

  public static void main(String[] args) {
```

```
46      System.out.println(new Retangulo(480, 300));
47      System.out.println(new Retangulo("480x300x100"));
48      System.out.println(new Retangulo("480q300"));
49      System.out.println(new Retangulo("480xqwe"));
50    }
51  }
```

O código presente entre as linhas 10 e 30 é usado para construir e validar a entrada. O método utilitário `ehInteiro(String):boolean` (entre as linhas 23 e 30) foi introduzido para verificar se é seguro executar o `parseInt`.

Seja com EAFP ou LBYL, Programadores Profissionais esperam e tratam situações excepcionais.

Eles tentam antever e tratar os problemas comuns, praticando a *Programação Defensiva.*

Exceções não são Erros se forem esperadas e tratadas

As exceções não, necessariamente, quebram o programa. Os programadores amadores normalmente veem as exceções como erros, enquanto os profissionais veem como uma situação previsível. A diferença é a experiência: os profissionais esperam sempre pelo pior.

Para fechar, considere o mesmo problema de contagem de palavras em Python usando LBYL:

LBYL em Python

```
1  def conta(frase):
2      palavras = frase.split()
3      contagem = {}
4      for word in palavras:
5          if word in contagem:
6              contagem[word] += 1
7          else:
8              contagem[word] = 1
9      return contagem
10
11 print conta("texto exemplo texto livro")
```

Nesse caso, o `if/else` entre as linhas 5 e 8 substituiu o `try/except`.

Será que opções demais atrapalham ou ajudam?

Quero dizer, se tu tens apenas um terno (digo a roupa) para situações especiais (eu tenho um só que uso em todas as formaturas :) é muito fácil vestir-se rápido. Mas com dois ou mais, a escolha é mais difícil, certo?[a]

Outro ditado popular é: *uma pessoa com um relógio sabe a hora exata, uma pessoa com dois relógios sabe a média aproximada.*

Ok, esses quadros têm só delongas, mas o que quero te dizer é para não ficares 3 horas pensando: *uso EAFP ou LBYL aqui?*. Existem poucos *tradeoffs* entre as duas técnicas, então pense sempre em escrever um código que não falha em primeiro lugar, seja com uma ou outra técnica.

[a] Uma palestra excelente sobre problemas de escolha é *"O paradoxo da escolha"* disponível no TED Talks em https://www.ted.com/talks/barry_schwartz_on_the_paradox_of_choice

Métodos fábrica estáticos

Métodos estáticos para fabricação (*static factory methods*) servem como uma alternativa aos construtores. Eles têm muitas vantagens: são mais inteligíveis, flexíveis, comunicam a intenção, permitem instanciar objetos covariantes (subclasses ou classes concretas que implementam uma interface) e podem até devolver uma instância já existente (permitem reciclar objetos e economizar memória).

Para implementar, cria-se um método estático público, responsável por construir (se necessário) e retornar o objeto. Opcionalmente, pode-se tornar o construtor privado, para obrigar o instanciamento através dos *métodos fábrica*.

Como exemplo, considere a construção de um objeto `Dado` onde seja possível escolher a quantidade de lados, primeiro usando a abordagem tradicional, usando construtores. Ver código a seguir:

Abordagem tradicional: usando construtores para instanciar objetos

```
1  package construtores;
2
3  public class Dado {
4
5    private int valor;
6    private final int lados;
7
8    public Dado() { this(6); }
9
10   public Dado(int lados) {
11       this.lados = lados;
12   }
13
14   public int getLados() {
15     return lados;
16   }
17
18   public void joga() {
19     this.valor = (int) (Math.random() * lados + 1);
```

```
20    }
21
22    public int getValor() {
23      return valor;
24    }
25
26    @Override
27    public String toString() {
28      return "Dado(lados:" + lados + ", valor:" + valor + ")";
29    }
30
31    public static void main(String[] args) {
32      Dado dado = new Dado();
33      for (int i = 0; i < 10; i++) {
34        dado.joga();
35        System.out.println(dado);
36      }
37      dado = new Dado(20);
38      for (int i = 0; i < 10; i++) {
39        dado.joga();
40        System.out.println(dado);
41      }
42    }
43  }
```

Sobre o código o anterior, considere as linhas 32 e 37. Presume-se que por padrão o dado tenha 6 lados e o parâmetro usado na construção seja para personalizar, mas são dois conceitos implícitos. O código-fonte pode ser refatorado para usar um *método fábrica estático* no lugar do construtor, tornando explícita a intenção:

Abordagem alternativa: usando método fábrica para instanciar objetos

```
1   package fabrica;
2
3   public class Dado {
4
5     private int valor;
6     private final int lados;
7
8     private Dado(int lados) {
9         this.lados = lados;
10    }
11
12    public static Dado lados(int lados) {
13      return new Dado(lados);
14    }
15
```

```
16    public static Dado seisLados() {
17      return new Dado(6);
18    }
19
20    public int getLados() {
21      return lados;
22    }
23
24    public void joga() {
25      this.valor = (int) (Math.random() * lados + 1);
26    }
27
28    public int getValor() {
29      return valor;
30    }
31
32    @Override
33    public String toString() {
34      return "Dado(lados:" + lados + ", valor:" + valor + ")";
35    }
36
37    public static void main(String[] args) {
38      Dado dado = Dado.seisLados();
39      for (int i = 0; i < 10; i++) {
40        dado.joga();
41        System.out.println(dado);
42      }
43      dado = Dado.lados(20);
44      for (int i = 0; i < 10; i++) {
45        dado.joga();
46        System.out.println(dado);
47      }
48    }
49  }
```

As alterações necessárias aparecem entre as linhas 8 e 18: tornar o construtor privado e introduzir métodos estáticos que retornam uma instância. O método na linha 16 predefine um dado de seis lados (poderiam ser predefinidos outros mais comuns). A principal vantagem dos *métodos fábrica estáticos* é que eles têm nomes, como pode ser visto nas linhas 38 e 43, isto é, `Dado.lados(20)` é mais expressivo que `new Dado(20)`.

Programadores Profissionais reconhecem que *explícito é melhor que implícito*.

Essa declaração está presente no Zen do Python, é um aforismo fantástico, entre muitos outros disponíveis no PEP 20: Zen of Python[39]. Substituir construtores por *métodos fábrica estáticos* é, muitas vezes, uma forma de tornar uma construção ofuscada/ambígua em visível/evidente.

O fato de *métodos fábrica estáticos* terem nome também ajuda na implementação de construções ambíguas, que não seriam implementáveis com construtores. Por exemplo, considere uma classe para representar *peso* em três unidades de medida: gramas, kilos e libras. O desafio está em permitir instanciar *peso* em qualquer uma dessas unidades, por exemplo, considere `new Peso(340)`, seriam 340gr, 340kg ou 340lb?

Peso.java

```
1  package construtores;
2
3  public class Peso {
4
5    private int gramas;
6
7    public Peso(int gramas) {
8      this.gramas = gramas;
9    }
10
11   public Peso(double kilos) {
12     this.gramas = (int) (kilos * 1000);
13   }
14
15   public Peso(double libras) {
16     this.gramas = (int) (libras / 0.00220462262);
17   }
18
19   public int getGramas() {
20     return this.gramas;
21   }
22
```

Sobrecarga impossível de construtores

Considerando o código da imagem anterior, é impossível implementar a construção com kilos e libras (o código não compila). O problema está na coincidência do tipo dos parâmetros - kilos e libras são do tipo `double`. Esse empasse pode ser solucionado com métodos estáticos para fabricar os pesos, como pode ser visto no código a seguir:

[39] https://www.python.org/dev/peps/pep-0020/

Usando métodos fábrica para contornar incompatibilidade de construtores

```
1  package fabrica;
2
3  public class Peso {
4
5    private int gramas;
6
7    private Peso(int gramas) {
8      this.gramas = gramas;
9    }
10
11   public static Peso emGramas(int gramas) {
12     return new Peso(gramas);
13   }
14
15   public static Peso emKilos(double kilos) {
16     return new Peso((int) (kilos * 1000));
17   }
18
19   public static Peso emLibras(double libras) {
20     return new Peso((int) (libras / 0.00220462262));
21   }
22
23   public int getGramas() {
24     return this.gramas;
25   }
```

Considerando o código, a implementação de métodos fábrica é relativamente simples. É recomendado proibir o uso do construtor tornado-o privado (linha 7) e adicionar quantos *métodos fábrica estáticos* forem necessários (linhas 11 a 21) para lidar com todas as possibilidades e facilidades de construção. Usando essa estrutura se torna muito fácil e claro instanciar peso de diferentes medidas, por exemplo, considere as situações de uso a seguir:

Usando métodos fábrica para contornar incompatibilidade de construtores

```
1    public static void main(String[] args) {
2      Peso p1 = Peso.emGramas(2300);
3      Peso p2 = Peso.emKilos(2300);
4      Peso p3 = Peso.emLibras(2300);
5    }
```

POG PATTERN

Sabe, conheço pessoas que fariam os construtores com *kilos* e *libras* dos tipos *double* e *float* para evitar incompatibilidade. Deste modo `new Peso(1000)` são mil gramas, `new Peso(1000.0)` ou `new Peso(1000D)` são mil kilos e `new Peso(1000F)` são mil libras. Como dá para perceber

> é difícil adivinhar qual construtor se aplica a cada medida. É um, entre muitos, Padrões POG (Programação Orientada a Gambiarras).

O uso de um *método estático para fabricação* também permite limitar a quantidade de objetos criados, controlando quando é necessário ou não fazer um `new`. Entre as aplicações está a economia de memória. Por exemplo, se for necessário o peso de 1kg seguidas vezes, apenas um objeto pode ser criado em vez de vários. Esta situação pode ser vista no código a seguir:

Usando métodos fábrica para reciclar objetos

```
1  package reciclagem;
2
3  public class Peso {
4
5    private int gramas;
6    private static Peso ultimoCriado = new Peso(0);
7
8    private Peso(int gramas) {
9      this.gramas = gramas;
10   }
11
12   public static Peso emGramas(int gramas) {
13     if (gramas != ultimoCriado.gramas) {
14       System.out.println("criando um peso de " + gramas + "g");
15       ultimoCriado = new Peso(gramas);
16     }
17     return ultimoCriado;
18   }
19
20   public static Peso emKilos(double kilos) {
21     return emGramas((int) (kilos * 1000));
22   }
23
24   public static Peso emLibras(double libras) {
25     return emGramas((int) (libras / 0.00220462262));
26   }
27
28   public int getGramas() {
29     return this.gramas;
30   }
31
32   public double getKilos() {
33     return this.gramas / 1000.0;
34   }
35
```

```
36    public double getLibras() {
37      return this.gramas * 0.00220462262;
38    }
39
40    public String toString() {
41      return this.gramas + "gr";
42    }
43
44    public static void main(String[] args) {
45      Peso p1 = Peso.emGramas(1000);
46      System.out.println(p1);
47      Peso p2 = Peso.emKilos(1);
48      System.out.println(p2);
49      Peso p3 = Peso.emLibras(2.205);
50      System.out.println(p3);
51    }
52  }
```

Sobre o código anterior, a reciclagem é implementada guardando o último objeto peso criado. Se for solicitado o mesmo peso novamente ele devolve o último, sem um novo `new` (redundante, eu sei :). A saída do método `main` é a seguinte:

```
criando um peso de 1000g
1000gr
1000gr
1000gr
```

Programadores Profissionais se preocupam com o uso de memória.

Os amadores não conhecem os detalhes do gerenciamento de memória e os negligentes confiam no mecanismo de gerenciamento e na quantidade exagerada de memória disponível nos computadores atuais. Já os Profissionais se preocupam com o uso da memória, eles sabem que ela acaba uma hora ou outra (cai uma lágrima ao ver um `OutOfMemoryError`).

Relançamento de exceção

O lançamento e captura de exceções é o *"jeito"* moderno de fazer a gestão de possíveis *"erros"* no programa (antigamente eram usados códigos de erro (e ainda são, pelos *relutantes*)). Entre as utilidades das exceções, destaca-se a possibilidade de rastreamento (talvez seja a funcionalidade mais importante) fazendo com que uma exceção, mais específica, seja representada por outra, mais genérica, em sequência até resultar em uma mensagem de erro para o usuário final (ou então fazer o programa *cair* – ver Fail-Fast no quadro de info a seguir).

Códigos de Erro

Retornar *códigos de erro* é uma técnica comum usada por programadores *old school*, advindos de C, por exemplo, e aplicável em qualquer linguagem, sendo uma solução mais leve. As exceções têm outros pontos positivos como a facilidade de depuração, pois permitem carregar mais informações sobre o problema.

Fail-Fast

Tu deves estar te perguntando o porquê de fazer seu programa cair. Muitas vezes é melhor derrubar o programa do que deixá-lo funcionando num estado inválido e acarretar mais problemas. Esse princípio é conhecido como *Fail Fast.*

As exceções podem (devem) ser tratadas. Se já programas em Java, é bem provável que tenhas lidado com exceções lançadas por classes do sistema (API). Algumas são obrigatórias (*checked exceptions*) que devem ser abraçadas com `try/catch` ou declaradas (repassadas). Quando as exceções não são *checked* (em outras linguagens, nenhuma é, como em C#) nos restam duas opções, profissionais, para lidar com elas:

1. **Tratar**: capturamos a exceção, verificamos o que está errado, tentamos novamente ou mostramos uma mensagem para o usuário.
2. **Relançar**: relançamos (repassamos) a exceção para ser tratada pelo método (e programador) chamador.

Contudo, existem outras opções, amadoras, que são contra-indicadas:

1. **Ignorar**: não capturamos, ignoramos a possibilidade de uma exceção acontecer e se acontecer *"o destino saberá o que fazer"*.
2. **Suprimir**: capturar a exceção e fazer nada com ela, prática conhecida também como *"engolir a exceção"*, causando uma falha silenciosa.
3. **Declarar**: declarar que o método lança exceção e deixar *"vazar"* para o método chamador (passar para frente).

"Engolir" as exceções é uma prática amadora.

Programadores amadores, por vezes, abraçam a instrução com `try/catch` e deixam o `catch` vazio, sem qualquer atitude em caso de exceção. É análogo a *"varrer a sujeira para debaixo do tapete"*.

Declarar (repassar) as exceções sem fazer qualquer tratamento pode ser contra-indicado.

Nem sempre declarar a exceção é uma boa opção. Se ela não for tratada em um método, deverá ser tratada em outro. Algumas vezes a exceção repassada é de responsabilidade da classe/método onde aconteceu, por exemplo, uma `SQLException` deve ser tratada na classe que lida com o banco de dados, pois se repassada pode ter de ser tratada em uma parte do sistema onde tem nada a ver com SQL, pense nisto.

Programadores Profissionais tratam e, às vezes, relançam.

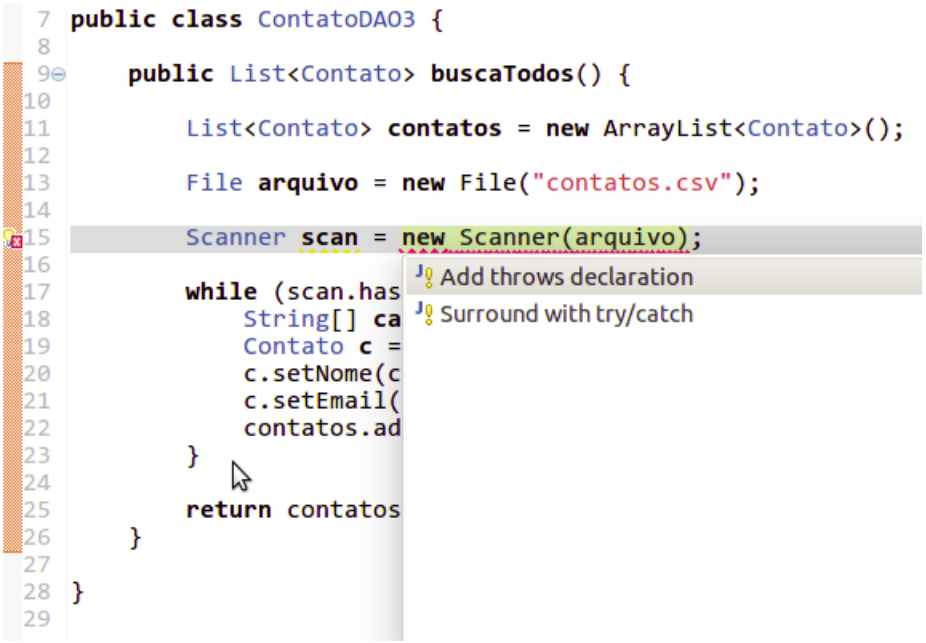
Eles não ignoram, eles não suprimem!

Para entender a motivação e o objetivo de relançar exceções, considere o seguinte exemplo: considere um pequeno aplicativo para ler contatos de uma fonte de dados, por exemplo, de um arquivo `CSV`. O código para ler os dados é encapsulado em um objeto de acesso a dados, que é um padrão comum para gravação e leitura de informações chamado DAO (Data Access Object). Esse exemplo pode ser visto no código a seguir:

Objeto de Acesso a Dados

```
7  public class ContatoDAO3 {
8
9    public List<Contato> buscaTodos() {
10
11     List<Contato> contatos = new ArrayList<Contato>();
12
13     File arquivo = new File("contatos.csv");
14
15     Scanner scan = new Scanner(arquivo);
16
17     while (scan.hasNextLine()) {
18       String[] campos = scan.nextLine().split(";");
19       Contato c = new Contato();
20       c.setNome(campos[0]);
21       c.setEmail(campos[1]);
22       contatos.add(c);
23     }
24
25     return contatos;
26   }
27
28 }
```

Existe um erro neste código que o impede de compilar, na linha 15. A linguagem Java usa exceções verificadas, que é a obrigação de ou tratar uma possível exceção ou de repassá-la. No Eclipse aparece a seguinte *hint* (dica) do *code assist* (assistente de código) na imagem a seguir:



Ou repassarás ou tratarás o_O

Conforme imagem anterior, a instrução `Scanner(File)` faz o Eclipse solicitar ação para `FileNotFoundException`. Neste instante, é possível é fazer o *surround* com `try/catch` (que significa tratar) ou adicionar um `throws` ao método (o que significa declarar). A solução simples é adicionar um `throws` ao método, que significa repassar a exceção para quem está chamando o método `buscaTodos`. Neste caso estamos *"vazando"* a exceção de um nível mais baixo (acesso a arquivo) para um nível mais alto (interface com o usuário, por exemplo). No futuro, uma alteração para ler os contatos de um Banco de Dados SQL, nos faria repassar uma `SQLException`, alterando a assinatura do nosso método e fazendo com que os dependentes desse método também tenham que lidar com a `SQLException`. É uma violação básica de Separação de Interesses, um princípio de arquitetura de sistemas.

Arquitetura e Separation of Concerns

Este livro não pretende adentrar o assunto de Arquitetura de Software, mas é importante compreender que durante o projeto de um sistema se pensa em blocos maiores que classes, como camadas, componentes, etc. Um princípio básico para fazer a separação dos módulos é conhecido como Separação de Interesses (Separation of Concerns - SOC, em inglês), que orienta a agrupar pacotes, classes, métodos, etc, segundo o *"tema"* que abordam, evitando que um interesse invada espaço do outro.

A segunda solução é tratar, ou seja, *abraçar* a instrução com `try/catch` para capturar a exceção, caso aconteça. O problema aqui é: após capturar a exceção o que fazemos com ela? Muitos programadores suprimem a exceção, por exemplo:

Abraçando instruções com try/catch

```
7  public class ContatoDAO2 {
8    public List<Contato> buscaTodos() {
9      List<Contato> contatos = new ArrayList<Contato>();
10     try {
11       File arquivo = new File("contatos.csv");
12       Scanner scan = new Scanner(arquivo);
13       while (scan.hasNextLine()) {
14         String[] campos = scan.nextLine().split(";");
15         Contato c = new Contato();
16         c.setNome(campos[0]);
17         c.setEmail(campos[1]);
18         contatos.add(c);
19       }
20     } catch (FileNotFoundException e) {
21       e.printStackTrace();
22     }
23     return contatos;
24   }
25 }
```

O problema está na linha 21, onde é feito o `e.printStackTrace()`. Esta instrução não faz com o que o programa *"caia"*, então o efeito é que os contatos não sejam carregados e isso acontece silenciosamente, ou seja, acabamos de suprimir uma exceção. Outras variantes dessa má prática são deixar o `catch` vazio `catch (FileNotFoundException e) {}` ou imprimir uma mensagem sem utilidade no console, como a seguir:

```
// ...
} catch (FileNotFoundException e) {
    System.out.println("aconteceu um erro");
}
// ...
```

O problema de capturar, mas não tratar as exceções!

Toda exceção capturada e não tratada adequadamente pode acabar escondendo falhas no programa.

Uma opção usada por programadores profissionais é o ***rethrow*** (relançamento da exceção). O relançamento é uma técnica útil para não deixar as exceções de um nível mais baixo de abstração vazarem (tais como: `IOException`, `FileNotFoundException`, `SQLException`) e ao mesmo tempo não deixar eventuais problemas passarem despercebidos e sem nenhum tipo de tratamento. A mecânica do *rethrow* é relativamente simples: capturar a exceção e *embrulhar* em uma nova exceção de abstração apropriada (`DAOException`, por exemplo). A seguir como fica o ContatoDAO usando esta técnica:

Relançando a exceção

```
7  public class ContatoDAO1 {
8    public List<Contato> buscaTodos() {
9      List<Contato> contatos = new ArrayList<Contato>();
10     try {
11       File arquivo = new File("contatos.csv");
12       Scanner scan = new Scanner(arquivo);
13       while (scan.hasNextLine()) {
14         String[] campos = scan.nextLine().split(";");
15         Contato c = new Contato();
16         c.setNome(campos[0]);
17         c.setEmail(campos[1]);
18         contatos.add(c);
19       }
20     } catch (FileNotFoundException e) {
21       System.err.println("Arquivo de dados não encontrado: " + e);
22       throw new RuntimeException("Fonte de dados não encontrada", e);
23     }
24     return contatos;
25   }
26 }
```

Na linha 21 a exceção específica `FileNotFoundException` é embrulhada (*wrapped*) em uma `RuntimeException`, que é mais genérica. Na prática, o programa vai cair (se não houver tratamento subsequente) e a exceção não passará despercebida. A vantagem está no tipo de exceção, quem chama o método `buscaTodos` não vai lidar diretamente com `FileNotFoundException`, aumentando a flexibilidade do subsistema, por exemplo, caso passemos a usar `SQLException` no futuro, quem chama nosso método `buscaTodos` não será afetado.

Daria para fazer um livro a parte apenas sobre tratamento de exceções e depuração, é um assunto muito rico, por vezes com conceitos mal compreendidos e/ou subutilizados. Entretanto, para fechar este tópico, vou apresentar a seguir um algoritmo aceitável na maioria das vezes para lidar com exceções:

- Capturar as exceções de nível mais baixo;
- Registrar (*logar*);
- Relançar num nível apropriado segundo a camada que se está trabalhando (neste caso, persistência).

Acompanhe o exemplo a seguir:

Registrando e relançando uma exceção no nível de abstração correto

```
8  public class ContatoDAO {
9    public List<Contato> buscaTodos() {
10     List<Contato> contatos = new ArrayList<Contato>();
11     Scanner scan = null;
12     try {
13       File arquivo = new File("contatos.csv");
14       scan = new Scanner(arquivo);
15       while (scan.hasNextLine()) {
16         String[] campos = scan.nextLine().split(";");
17         Contato c = new Contato();
18         c.setNome(campos[0]);
19         c.setEmail(campos[1]);
20         contatos.add(c);
21       }
22     } catch (FileNotFoundException e) {
23       Logger.getLogger("ContatoDAO").warning(e.toString());
24       throw new DAOException("Fonte de dados não encontrada", e);
25     } finally {
26       if (scan != null)
27         scan.close();
28     }
29     return contatos;
30   }
31 }
```

Na linha 24 a exceção é registrada e na linha 25 ela é relançada como uma DAOException, que é uma exceção consistente e com o nível correto de abstração, digo, quem chama métodos de uma classe DAO pode esperar por `DAOException`'s se algo der errado.

Programadores Profissionais avaliam os níveis de abstração.

Eles planejam o lançamento e tratamento adequado para cada camada, implementam novas exceções, em níveis que vão dos mais gerais aos mais específicos.

Excessivos lançamentos e relançamentos de exceções depreciam a performance.

Como tudo, usar exceções tem um *trade-off*. O lançamento e tratamento frequente de exceções implica na criação de (novos) objetos, desvios de fluxo e outros detalhes, incorrendo em depreciamento, embora de leve, da performance.

Concatenação de *strings* com *builders* e *appenders*

Tu podes estar te perguntando o porquê de existir uma técnica especial para concatenar *strings*. A situação é que concatenar *strings*, em muitas linguagens e incluindo Java, tem uma performance bem baixa, além de desperdiçar muita memória. Isso acontece devido ao projeto da classe `String` como um tipo imutável (é o caso em Java), então cada concatenação gera uma nova instância de `String`. Para exemplificar o problema considere o seguinte código:

```
String s1 = "teste";
s1 = s1 + ", testando, " + s1 + ", som, 1, 2, 3, " + s1;
```

A pergunta é: quantos objetos `String` foram criados? Tempoooo, tic, tac, tic, tac, resposta: 5. Cada vez que é usado o operador + é criada uma nova instância, resultado da concatenação de duas *strings*. Na prática, o código anterior gera as seguintes *strings*:

```
"teste"
"teste" + ", testando, " = "teste, testando, "
"teste, testando, " + "teste" = "teste, testando, teste"
"teste, testando, teste" + ", som, 1, 2, 3, " = "teste, testando, teste, som, 1, \
2, 3, "
"teste, testando, teste, som, 1, 2, 3, " + "teste" = "teste, testando, teste, som\
, 1, 2, 3, teste"
```

Este comportamento de `String` causa perda de desempenho e muito uso de memória quando a operação de concatenação é executada muitas vezes (em loops e, por exemplo, em uma aplicação *web* ou *web service*). Para avaliar, vamos medir o desempenho fazendo um teste no código a seguir:

Medindo o desempenho da concatenação de strings em Java

```
1  public class Main2 {
2    public static void main(String[] args) {
3      long inicio = System.currentTimeMillis();
4      String s1 = "teste";
5      for (int i = 0; i < 10000; i++) {
6        s1 = s1 + ", testando, ";
7      }
8      long fim = System.currentTimeMillis();
9      System.out.println((fim - inicio) + "ms");
10   }
11 }
```

No meu modesto *laptop* a operação durou `2100ms`, ou seja, um pouco mais de 2 segundos para executar dez mil concatenações. As instruções presentes nas linhas 3, 8 e 9 são muito úteis para fazer pequenos *benchmarks* (aferições de performance).

É importante esclarecer que não é uma característica apenas da linguagem Java. Para mostrar que o projeto é o mesmo em outras linguagens, vou mostrar a seguir o mesmo exemplo em Ruby:

Medindo o desempenho da concatenação de strings em Ruby

```ruby
inicio = Time.now
s1 = "teste";
10000.times do
  s1 = s1 + ", testando, "
end
fim = Time.now
puts ((fim - inicio) * 1000).to_s + "ms"
```

O tempo médio foi de 440ms para dez mil concatenações, excelente resultado se comparado com Java, mas que pode ficar melhor usando a técnica desde tópico.

Sempre culpar a linguagem pelos problemas é uma atitude amadora.

Os profissionais buscam alternativas, meios, técnicas, padrões, até *hacks*, para fazer o melhor. Se nada der certo, eles até trocam de linguagem, mas não *"queimam"* a linguagem anterior, pois sabem que cada linguagem pode ter resultados melhores ou piores com diferentes grupos de problemas.

O problema de baixa performance na concatenação de *strings* geralmente é muito simples de ser solucionado, pois as linguagens oferecem estratégias mais eficientes de concatenação que devem ser usadas para execuções frequentes. Na linguagem Java é usada a classe `StringBuilder`, que disponibiliza o método `append` (acrescentar). Na prática, `StringBuilder` possui internamente um *array* de caracteres onde ele vai *"encostando"* os novos caracteres à direita, em vez de instanciar novos objetos *string*. A seguir está o mesmo teste de desempenho, só que agora com `StringBuilder`:

Concatenando do jeito certo com StringBuilder

```java
public class Main1 {
  public static void main(String[] args) {
    long inicio = System.currentTimeMillis();
    StringBuilder s1 = new StringBuilder("teste");
    for (int i = 0; i < 10000; i++) {
      s1.append(", testando, ");
    }
    String s2 = s1.toString(); // instancia a string resultante dos appends
    long fim = System.currentTimeMillis();
    System.out.println((fim - inicio) + "ms");

  }
}
```

O tempo médio foi de 5ms. Concordamos que bem melhor que os 2100ms quando concatenado da maneira tradicional (com o operador +)?

Programadores Profissionais conhecem a API da linguagem, eles são proficientes nela.

Eles sabem quais classes e métodos oferecem o melhor resultado. Programadores Java, por exemplo, usam `StringBuilder` para concatenar *strings* em sequência.

Para fechar o tópico, vamos resolver o problema também em Ruby. Nesta linguagem há um método especial `<<` (*append*), que realiza a mesma tarefa: acrescentar caracteres à direita em vez de instanciar uma *string* nova. A seguir o `benchmark` da solução em Ruby:

Concatenando strings do jeito certo em Ruby

```
1  inicio = Time.now
2  s1 = "teste";
3  10000.times do
4    s1 << ", testando, "
5  end
6  fim = Time.now
7  puts ((fim - inicio) * 1000).to_s + "ms"
```

O tempo baixou de `440ms` para `5ms`. Moral da história: use *builders*, *appenders*, ou outro recurso para *"crescer" strings* em vez do operador `+`.

Comparação na mão-esquerda

Existe uma falha, bem comum, que pode acontecer nas expressões booleanas onde os objetos são comparados por igualdade. Um descuido na construção dessas expressões (como em `if`s, `for`'s ou `while`'s) pode causar uma `NullPointerException` e até introduzir *bugs* difíceis de encontrar. O problema está na atribuição involuntária de uma variável quando se desejava apenas comparar.

O risco é ainda maior em linguagens que, diferente de Java, avaliam *verdadeiro/falso* também com números, nulos e listas, por exemplo, enquanto Java aceita apenas o tipo booleano `true` ou `false`, a linguagem C aceita números como `0` (para `false`) e `1` (um não-zero para `true`). Para ilustrar a situação em C, considere o seguinte código:

Pequeno código em C com expressões condicionais

```
1  #include <stdio.h>
2  int main(void) {
3    int x;
4    printf("Digite a opcao 1, 2 ou 3: ");
5    scanf("%d",&x);
6    if (x == 1) printf("primeira opcao\n");
7    if (x == 2) printf("segunda opcao\n");
8    if (x == 3) printf("terceira opcao\n");
9    printf("Opcao selecionada %d \n", x);
10 }
```

O código demonstrado no exemplo anterior funciona perfeitamente, por exemplo, se a entrada for 2 ele apresenta a seguinte saída:

```
Digite a opcao 1, 2 ou 3: 2
segunda opcao
Opcao selecionada 2
```

O exemplo é bobo, eu sei, mas a proposta aqui é deixar simples para explicar um *bug* igualmente bobo e sútil (o capiroto está nos detalhes). Considere que usemos apenas um = (em vez de ==) em um dos *if*'s, digamos em `if (x = 2)`. Este código é perfeitamente válido em C/C++, ele compila e executa, mas apresenta comportamento errático. O uso de um = atribui o valor 2 a variável `x` e avalia a expressão como verdadeiro (*não-0*). A seguir o código que apresenta tal problema:

Código em C com problema na expressão condicional

```
1  #include <stdio.h>
2  int main(void) {
3    int x;
4    printf("Digite a opcao 1, 2 ou 3: ");
5    scanf("%d",&x);
6    if (x == 1) printf("primeira opcao\n");
7    if (x = 2) printf("segunda opcao\n");
8    if (x == 3) printf("terceira opcao\n");
9    printf("Opcao selecionada %d \n", x);
10 }
```

Aquele `if` da linha 7 esconde um problema muito discreto, que passa pela fase de compilação (ou seja, não há erro no código). Este código apresenta a seguinte saída quando se digita 1, 2 e 3:

Saída inesperada

```
Digite a opcao 1, 2 ou 3: 1
primeira opcao
segunda opcao
Opcao selecionada 2

Digite a opcao 1, 2 ou 3: 2
segunda opcao
Opcao selecionada 2

Digite a opcao 1, 2 ou 3: 3
segunda opcao
Opcao selecionada 2
```

Perceba que há uma saída difícil de interpretar. Os *bugs* como esse são muito difíceis de resolver, por fazer o programa se comportar inesperadamente sem apresentar um erro explícito, ou seja, é uma *falha que não causa a quebra.*

Achar que as falhas existem apenas quando o programa cai é um pré-conceito amador.

O pior tipo de falha é a que não faz o programa cair. Muitas pessoas associam a palavra *"erro"* a um problema de compilação ou uma exceção durante a execução, etc, mas não a uma resposta incorreta, um comportamento errático. Lembre, há mais valor para o programador (e cliente, embora ele não saiba) quando o programa cai, em vez de funcionar de forma incorreta.

A técnica deste tópico chama-se *comparação na mão-esquerda* e tem o objetivo de tornar um problema implícito em explícito, ou seja, fazer a falha aparecer. Comparações do tipo `variavel == constante` são conhecidas como *comparação na mão-direita*. A comparação na mão-esquerda faz, claro, o inverso: `constante == variável`. Com esta técnica, quando se esquece um = no par ==, o programa passa a não compilar, o que é perfeito para nós (por mais incrível que pareça). Considere o exemplo:

Código em C com expressões condicionais escritas com comparação na mão-esquerda

```c
#include <stdio.h>
int main(void) {
  int x;
  printf("Digite a opcao 1, 2 ou 3: ");
  scanf("%d",&x);
  if (1 == x) printf("primeira opcao\n");
  if (2 = x)  printf("segunda opcao\n");
  if (3 == x) printf("terceira opcao\n");
  printf("Opcao selecionada %d \n", x);
}
```

A inversão das posições `constante <=> variável` não alteram a lógica condicional e trazem a vantagem de não compilar quando esquecemos um =. Esse é mais um caso onde se faz mais esforço para entender a motivação por trás, do que para usar a técnica em si.

Fechando o tópico, outras linguagens *"sofrem"* do mesma vulnerabilidade. Por exemplo, está a seguir o mesmo problema em PHP:

Código em PHP com problema na expressão condicional

```php
<?php
$x = readline("Digite a opcao 1, 2 ou 3: ");
if ($x == 1) print("primeira opcao\n");
if ($x = 2) print("segunda opcao\n");
if ($x == 3) print("terceira opcao\n");
print("Opcao selecionada $x \n");
```

O código anterior não apresenta erros, ele apenas se comporta mal, como apresentado no bloco *saída inesperada* visto anteriormente. O uso de *comparação na mão-esquerda* torna o código mais depurável, tornando o problema explícito. Considere o seguinte código PHP que usa esta técnica:

Código em PHP com expressões condicionais escritas com comparação na mão-esquerda

```
1  <?php
2  $x = readline("Digite a opcao 1, 2 ou 3: ");
3  if (1 == $x) print("primeira opcao\n");
4  if (2 = $x) print("segunda opcao\n");
5  if (3 == $x) print("terceira opcao\n");
6  print("Opcao selecionada $x \n");
```

Considere, agora, a saída produzida pelo código anterior:

Erros explícitos

```
PHP Parse error:  syntax error, unexpected '=' in comparacao-solucao.php on line 4
```

Na linguagem Java, adotada como padrão para este livro, as expressões condicionais avaliam apenas o tipo `boolean`, não apresentando exatamente o mesmo problema, porém um problema da mesma categoria usando o método `equals`: a expressão `variavel.equals(constante)` pode causar uma `NullPointerException` se a variável está `null`. Então, a mesma técnica pode ser usada para evitar o problema, usando *comparação na mão-esquerda*: `constante.equals(variavel)`. Para exemplificar, veja a seguir um código em Java que apresenta esse problema:

Mesmo problema em Java

```
1  package problema;
2
3  public class Main {
4    public static void main(String[] args) {
5      // String s = "teste";
6      String s = null;
7      if (s.equals("teste")) {
8        System.out.println("Ate aqui tudo bem :)");
9      }
10   }
11 }
```

O código anterior lança uma `NullPointerException` quando `s` é `null`. Tu podes **defender** usando a *comparação na mão-esquerda*: `"teste".equals(s)`, pois, em Java, qualquer objeto comparado com a ausência de valor (`null`) é avaliado como `false`. A seguir o mesmo código resolvido:

<<Mesma solução em Java[40]

Programadores Profissionais codificam defensivamente.

Eles tentam antever os problemas e procuram escrever códigos menos propensos a falhas.

[40] code/comparacao-mao-esquerda/src/solucao/Main.java

Programação Defensiva

Os códigos vistos neste tópico servem como exemplo de Programação Defensiva, uma técnica (ou prática) comum usada por programadores experientes para evitar *bugs* no programa.

Retorno cedo!

O *retorno cedo* (*Early Return*) é uma técnica utilizada para sair o mais brevemente possível de um método, eliminando processamento adicional. A principl vantagem de *retornar cedo* é a redução de expressões condicionais aninhadas (`if`'s dentro de `if`'s) e de indentação, o que melhora bastante a legibilidade dos métodos.

Retornar cedo e a performance

Sair de um método logo após ter o valor de retorno disponível sem executar as linhas subsequentes também pode trazer ganhos de performance em alguns casos.

Como ponto fraco, em métodos muito longos, por exemplo, com 50[41] linhas de código ou mais, o uso de múltiplos `returns` pode dificultar a depuração (eliminação de *bugs*), já que torna difícil saber, ao certo, qual ponto de saída está sendo executado e até antever o estado final do método, pois algumas linhas nem sempre são executadas.

A implementação do *retorno cedo* não é complicada, por exemplo, em métodos que executam determinadas instruções baseadas condicionalmente na entrada, substitua a variável que armazena o retorno por um `return` explícito. Já em métodos `void`, nas cláusulas-guarda, substitua a *condição para execução* por uma *condição para não-execução*, usando um `return;` vazio. Os dois casos serão mostrados nos exemplos a seguir.

Considere uma classe `Seguranca`, que tem o objetivo de definir o nível de acesso de um usuário segundo algumas de suas propriedades. Primeiro observe o exemplo a seguir, onde não é usado o *retorno cedo*:

Método que não aplica o *retornar cedo*

```
public class Seguranca {
  public static NivelAcesso privilegio(Usuario usuario) {
    NivelAcesso nivel = null;
    if (!usuario.isBloqueado()) {
      if (usuario.getCargo() != null) {
        if (usuario.getCargo() == Cargo.Gerente) {
          nivel = NivelAcesso.Administrador;
        } else if (usuario.getCargo() == Cargo.Funcionario) {
          nivel = NivelAcesso.Operador;
        }
```

[41] na verdade, um método com muitas linhas também não é uma boa prática, qualquer método com dezenas de linhas pode ser "*quebrado*" em métodos de menor tamanho.

```
      } else {
        nivel = NivelAcesso.Visitante;
      }
    } else {
      nivel = NivelAcesso.Nenhum;
    }
    return nivel; // único retorno do método
  }
```

O código do exemplo anterior obedece o *Only One Return* (único ponto de saída, que contrasta com o *Early Return*), mas ao custo do aninhamento de expressões condicionais e da existência de uma variável temporária `nivel`. Este método pode ser escrito usando o *retorno cedo*, fazendo com que cada condição declare um retorno imediato. A seguir, a mesma classe com as alterações para implementar o *retorno cedo*:

Método que *retorna cedo*

```
public class Seguranca {
  public static NivelAcesso privilegio(Usuario usuario) {
    if (usuario.isBloqueado()) {
      return NivelAcesso.Nenhum;
    }
    if (Cargo.Gerente.equals(usuario.getCargo())) {
      return NivelAcesso.Administrador;
    }
    if (Cargo.Funcionario.equals(usuario.getCargo())) {
      return NivelAcesso.Operador;
    }
    return NivelAcesso.Visitante;
  }
```

Comparando os dois exemplos anteriores, percebe-se que, além dos múltiplos `returns`, a lógica do método foi redesenhada. Entretanto, ambos atendem os requisitos e passam nos testes. A vantagem está na redução do aninhamento/deslocamento. A seguir outro exemplo, com o objetivo de simplificar as cláusulas-guarda:

Cláusulas salva-guarda sem *retornar cedo*

```
  public static void geraSenha(Usuario usuario) {
    if (usuario != null) {
      if (!usuario.isBloqueado()) {
        usuario.setSenha(java.util.UUID.randomUUID().toString().split("-")[0]);
      }
    }
  }
```

Cláusulas salva-guarda que *retornam cedo*

```
public static void geraSenha(Usuario usuario) {
  if (usuario == null || usuario.isBloqueado()) {
    return; // retorna sem gerar a senha
  }
  usuario.setSenha(java.util.UUID.randomUUID().toString().split("-")[0]);
}
```

Nos exemplos, considera-se um método `geraSenha` com as restrições de o usuário não ser nulo e não estar bloqueado. O primeiro exemplo *guarda* a instrução com dois `if`'s aninhados (poderia ser usado um `&&`). No segundo exemplo, a *guarda* faz a condição de **não-execução** para sair antecipadamente – isto é, antes de gerar a senha.

Retorno booleano

Esta técnica é bem simples e se baseia nos métodos que são projetados para retornar `true` ou `false` baseados em condições simples geralmente codificadas com `if/else`. Para exemplificar o problema considere o seguinte código:

Implementação inexperiente de retorno boolean

```
3  public class NumberUtil {
4
5          public static boolean ehImpar(int numero) {
6                  if (numero % 2 == 0) {
7                          return false;
8                  } else {
9                          return true;
10                 }
11         }
12
13         public static boolean ehNegativo(int numero) {
14                 if (numero < 0) {
15                         return true;
16                 } else {
17                         return false;
18                 }
19         }
```

É possível escrever os mesmos métodos sem `if/else`. Por exemplo, `numero < 0` na linha 12 representa uma expressão booleana, ou seja, ela é avaliada como `true` ou `false`. A eliminação dos `if/else` é simples: basta retornar a expressão que está dentro do `if`, que já é booleana e elegível para retorno. A seguir o código do exemplo anterior refatorado:

Retornando a expressão booleana

```
3  public class NumberUtil {
4
5          public static boolean ehImpar(int numero) {
6                  return numero % 2 != 0;
7          }
8
9          public static boolean ehNegativo(int numero) {
10                 return numero < 0;
11         }
```

Como vantagem, o código fica mais enxuto. Já foi discutido que nem sempre é favorável o código ser pequeno, mas neste caso especial pequeno é melhor.

Para finalizar este tópico, vamos para outro exemplo onde o *retorno booleano* pode ser aplicado, agora junto com a técnica de retornar cedo. A seguir está o código a ser refatorado:

Retorno booleano no fim do método

```
21         public static boolean ehNumero(String str) {
22                 int numeros = 0;
23                 for (char c : str.toCharArray()) {
24                         if (Character.isDigit(c)) {
25                                 numeros++;
26                         }
27                 }
28                 if (numeros == str.length()) {
29                         return true;
30                 } else {
31                         return false;
32                 }
33         }
```

O código anterior mostra uma situação onde o *retorno booleano* poderia ser usado junto com a técnica de *retornar cedo*. Neste caso em particular, temos um ganho de performance, pois é possível fazer o retorno, ou seja, dizer se a *string* é numérica ou não, sem precisar percorrer a *string* inteira todas as vezes. A seguir está o código refatorado:

Retorno booleano no fim do método

```
13          public static boolean ehNumero(String str) {
14                  for (char c : str.toCharArray()) {
15                          if ( ! Character.isDigit(c)) {
16                                  return false;
17                          }
18                  }
19                  return true;
20          }
```

Valor padrão e opcionalidade de parâmetros

A linguagem Java, pelo menos até a última versão (Java 8, enquanto escrevo), não disponibiliza uma sintaxe especial para atribuir valores padrão (*defaults*) para parâmetros opcionais e para variáveis locais não inicializadas. No entanto dá para contornar a falta desse recurso com duas técnicas: sobrecarga e operadores ternários.

Vamos entender o primeiro problema: a questão da *opcionalidade de parâmetros*. Considere um método que receba uma *string* com uma frase e um caractere separador, então os espaços são substituídos pelo caractere separador. Por exemplo, `Joiner.join("um teste", "*")` devolve `"um*teste"`. Agora, considere que informar o separador é opcional, por exemplo, `Joiner.join("um teste")` devolve `"um_teste"`, usando o *underline* _ (ou *underscore*) como padrão. Este tipo de recurso é conhecido como *opcionalidade de parâmetro* (ou argumento). Algumas linguagens suportam esse recurso através de atribuição na assinatura do método (ou função). Considere a seguinte implementação em C#:

Opcionalidade de parâmetros com atribuição padrão

```
1  public static class Joiner
2  {
3          public static string join(string entrada, string separador = "_") {
4                  return entrada.Replace (" ", separador);
5          }
6          static void Main ()
7          {
8                  System.Console.WriteLine(Joiner.join("um teste em csharp"));
9          }
10 }
```

O método apresentado pode ser invocado `Joiner.join("um teste em csharp")` sem passar o _ como segundo argumento. O valor padrão do parâmetro `separador` é um _, como pode ser visto na linha 3 `string separador = "_"`.

Na linguagem Java esse comportamento pode ser obtido com sobrecarga de métodos. A mecânica básica é criar um método que solicite todos os argumentos necessários, para então criar métodos que omitem esses argumentos e delegam a chamada para o método completo. Veja a implementação a seguir:

Opcionalidade de parâmetros com sobrecarga de métodos

```
1  public class Joiner {
2
3    public static String join(String entrada) {
4      return join(entrada, "_");
5    }
6
7    public static String join(String entrada, String separador) {
8      return entrada.replace(" ", separador);
9    }
```

A instrução chave do procedimento está na linha 4, onde o método delega a chamada passando o valor padrão (o *default*) _ para como argumento `separador` para o método completo.

Agora vamos entender o segundo problema: valor padrão para variáveis. É comum usar este recurso quando recebemos um parâmetro `null` ou que não é válido e não pretendemos deixar o programa cair. Considere que nosso `Joiner` receba os dois parâmetros como argumentos na linha de comando. Por exemplo, executando `java joiner "um teste" "*"` devolve `"um*teste"` e executando `java joiner "um teste"` devolve `"um-teste"`, usando hífen como valor padrão. Em Java essa falta do segundo argumento pode ser tratada de três formas:

- Com um `if`: é a mais simples (ou simplória) sendo a primeira solução que vem a cabeça dos iniciantes;
- Com a Técnica EAFP: tenta-se ler o segundo parâmetro, em caso de exceção usa o *default*;
- Com Operador Ternário: na verdade é um `if` em linha que devolve um valor – é muito útil para esses casos em particular.

As três abordagens aparecem no código a seguir:

Usando if, EAFP e Operador Ternário para implementar valor padrão

```
11   public static void main(String[] args) {
12     // usando IF para atribuir
13     // valor padrão na falta do segundo argumento
14     String separador = "-";
15     if (args.length > 1) separador = args[1];
16     System.out.println(join(args[0], separador));
17
18     // usando EAFP atribuir
19     // valor padrão na falta do segundo argumento
20     try {
21       separador = args[1];
22     } catch(Exception e) {
23       separador = "-";
24     }
25     System.out.println(join(args[0], separador));
```

```
26
27      // usando Operador Ternário para atribuir
28      // valor padrão na falta do segundo argumento
29      separador = args.length > 1 ? args[1] : "-";
30      System.out.println(join(args[0], separador));
31    }
32  }
```

O trecho entre a linha 20 e 25 é uma aplicação da técnica EAFP. Os outros dois usam a técnica LBYL, já que verificam a existência do parâmetro antes de ler. As linhas 29 e 30 mostram o uso de Operador Ternário, fazendo a mesma lógica do `if`, mas com uma atribuição direta. Uma das vantagens do Operador Ternário é não ter de pré-atribuir, ou seja, usado apenas o ternário a declaração e inicialização poderiam ficar na mesma linha, por exemplo: `String entrada = args.length > 1 ? args[1] : "-";`.

Só para fechar, outras linguagens possuem sintaxe específica para esses fins. Por exemplo, em Ruby, a opcionalidade de parâmetros e atribuição de valor padrão são implementados da seguinte maneira:

Mesmo problema solucionado em Ruby

```
1  def join(texto, separador = '_')
2    texto.gsub(' ', separador)
3  end
4
5  e = ARGV[0]
6  s = ARGV[1] || "-"
7  puts join e, s
```

Ruby implementa opcionalidade atribuindo um valor padrão para o parâmetro declarado na assinatura do método, como pode ser visto na linha 1 `separador = '_'`. Na linha 6 aparece o uso de dois *pipes* (o símbolo `|` lê-se *"páipi"*) que servem para atribuir um valor padrão caso o primeiro valor avalie como falso (em Ruby um índice não existente em uma lista retorna `nil` (`nil` do Ruby é o mesmo `null` de Java (chega de parênteses? :))) – em JavaScript também.

Operador ELVIS ?:

Algumas linguagens disponibilizam operadores especiais para a operação conhecida como *null coalescing*. O mais popular é chamado de *Elvis Operator*, sendo escrito com o símbolo `?:` (que lembra o topete do Elvis – ver emoticons – pré-emoji). Ele é muito útil para atribuir valores padrão, simplificando expressões ternárias. Por exemplo, considere o seguinte código em Java:

```
String param = null;
String s = param != null ? param : "padrão";
System.out.println(s); // imprime "padrão"
```

O propósito do código anterior é usar o valor de `param` se houver. Escrito em Kotlin, uma linguagem alternativa mas compatível com Java, aceita como padrão pela Google para desenvolvimento Android e que suporta o *Elvis Operator*, o código do exemplo anterior ficaria assim:

```
var param = null
var s = param ?: "padrão"
println(s) // imprime "padrão"
```

Muito mais enxuto. Entre as linguagens que suportam *Elvis Operator* estão Groovy, Kotlin, PHP e outras, às vezes com variações, em Swift, por exemplo, o símbolo é `??`.

Ordenar expressões condicionais do menor ao maior custo

Este tópico é para fechar a seção de técnicas trazendo uma questão de performance. Na verdade é um questão de *micro otimização* (ganhar *"um pouquinho"* de performance). A intenção é organizar instruções classificando-as do menor ao maior custo computacional (*custo computacional* quer dizer maior uso de memória e/ou processamento).

Para esclarecer, considere o seguinte exemplo de uma classe `UploadManager` (gerente de *uploads*), que seria usada num Servidor Web para gerenciar arquivos que os usuários *"sobem"* no servidor, como fotos, vídeos, etc. Sua função é permitir o *upload*, mas com um limite de quantidade de arquivos e tamanho de cada arquivo. Confira o código a seguir:

Classe UploadManager

```
7  public class UploadManager {
8
9    private final File dir;
10   private final long maxArquivos;
11   private final long maxTamanho;
12
13   public UploadManager(String dir, long maxArquivos,
14       long maxTamanho) {
15     this.dir = new File(dir);
```

```
16      this.maxArquivos = maxArquivos;
17      this.maxTamanho = maxTamanho;
18    }
19
20    public boolean store(File arquivo) {
21      if (totalFilesInDir(dir) < maxArquivos
22          && arquivo.isFile()
23          && arquivo.length() < maxTamanho) {
24        try {
25          FileOutputStream fos =
26              new FileOutputStream(dir.getAbsolutePath() + "/"
27              + UUID.randomUUID().toString());
28          FileInputStream fis = new FileInputStream(arquivo);
29          int b = -1;
30          while ((b = fis.read()) != -1) {
31            fos.write(b);
32          }
33          fis.close();
34          fos.flush();
35          fos.close();
36          return true;
37        } catch (Exception e) {
38          throw new RuntimeException("nao foi "
39              + "possivel armazenar", e);
40        }
41      } else {
42        return false;
43      }
44    }
45
46    public int totalFilesInDir(File dir) {
47      File[] listFiles = dir.listFiles();
48      if (listFiles == null) {
49        return 0;
50      }
51      int total = 0;
52      for (File f : listFiles) {
53        if (f.isFile()) {
54          total++;
55        }
56        if (f.isDirectory()) {
57          total += totalFilesInDir(f);
58        }
59      }
60      return total;
61    }
```

```
62
63    public static void main(String[] args) {
64      UploadManager um =
65          new UploadManager("/home/marcio/uploads",
66              1000000L, 4000000L);
67      System.out.println(
68          um.store(new File("/home/marcio/arquivo.doc")));
69    }
70
71  }
```

Para entender o exemplo preste atenção no código entre as linhas 21 e 23. O `if` na linha 21 verifica se o arquivo pode ser armazenado. Existem três condições neste `if`: `totalFilesInDir(dir) < maxArquivos && arquivo.isFile() && arquivo.length() < maxTamanho`. Qual dessas instruções tu achas que tem o maior custo? Difícil prever não é? Mas num experimento que fiz, usando um diretório com muitos arquivos, a instrução `totalFilesInDir` demorou alguns segundos. Esta técnica propõe deixar essa instrução por último na expressão condicional.

Para entender a solução, é importante entender como funcionam os operadores `&` e `&&` (e também o `|` e `||`). O operador `&` avalia as duas expressões – assim como o `|`. Já o operador `&&` avalia a próxima expressão apenas se a primeira for *true*. Se a primeira expressão for *false* não há necessidade de avaliar a próxima expressão mesmo que seja *true*, pois a expressão inteira já será *false*. No caso de `||`, a próxima instrução é avaliada quando as anteriores foram *false*, pois se uma instrução for *true* toda a expressão é *true*.

Operadores `&&` e `||` são conhecidos como *operadores de curto-circuíto*

Eles tem um comportamento diferente do `&` e `|`, que são conhecidos como `and` e `or` lógicos.

Grandes poderes trazem grandes responsabilidades não é mesmo? Com esse conhecimento percebe-se que `a && b` pode ter diferença com `b && a`. O objetivo agora é fazer com que as instruções mais leves e/ou mais prováveis de serem *true* sejam avaliadas primeiro nas instruções *if*, enquanto as instruções mais pesadas e com menor probabilidade sejam avaliadas mais para o fim. A seguir o código usado no exemplo anterior, agora otimizado:

Classe UploadManager com a instrução condicional refatorada

```
7   public class UploadManager {
8
9     private final File dir;
10    private final long maxArquivos;
11    private final long maxTamanho;
12
13    public UploadManager(String dir, long maxArquivos,
14        long maxTamanho) {
15      this.dir = new File(dir);
```

```
16      this.maxArquivos = maxArquivos;
17      this.maxTamanho = maxTamanho;
18    }
19
20    public boolean store(File arquivo) {
21      if (arquivo.isFile()
22          && arquivo.length() < maxTamanho
23          && totalFilesInDir(dir) < maxArquivos) {
24        try {
25          FileOutputStream fos =
26              new FileOutputStream(dir.getAbsolutePath() + "/"
27              + UUID.randomUUID().toString());
28          FileInputStream fis = new FileInputStream(arquivo);
29          int b = -1;
30          while ((b = fis.read()) != -1) {
31            fos.write(b);
32          }
33          fis.close();
34          fos.flush();
35          fos.close();
36          return true;
37        } catch (Exception e) {
38          throw new RuntimeException("nao foi "
39              + "possivel armazenar", e);
40        }
41      } else {
42        return false;
43      }
44    }
45
46    public int totalFilesInDir(File dir) {
47      File[] listFiles = dir.listFiles();
48      if (listFiles == null) {
49        return 0;
50      }
51      int total = 0;
52      for (File f : listFiles) {
53        if (f.isFile()) {
54          total++;
55        }
56        if (f.isDirectory()) {
57          total += totalFilesInDir(f);
58        }
59      }
60      return total;
61    }
```

```
62
63    public static void main(String[] args) {
64      UploadManager um =
65          new UploadManager("/home/marcio/uploads",
66              1000000L, 4000000L);
67      System.out.println(
68          um.store(new File("/home/marcio/arquivo.doc")));
69    }
70
71  }
```

A melhoria no código está entre as linhas 21 e 23. A verificação `totalFilesInDir(dir)` foi movida para o fim da expressão. Neste caso se `arquivo.isFile()` for *false* não será verificado o tamanho do arquivo e a quantidade de arquivos existentes, pois não é necessário, economizando recursos (CPU, memória, etc).

Existem muitas práticas relacionadas a estrutura de expressões condicionais.

O suficiente para criar um livro inteiro sobre o tema. Fica para uma próxima publicação ;)

Considerações sobre técnicas

As técnicas fazem diferença. Tanto a presença quanto a falta delas são notáveis. Aprender a escrever códigos com mais qualidade técnica deve ser uma das primeiras preocupações dos aprendizes, pois o código é o resultado do seu trabalho (*mostre-me teu código e te direi quem és*) tal como uma pintura, uma música, e o assentamento de porcelanato (digo isso por que vejo alguns desalinhados e desnivelados na minha sala `o_O`).

Todo programador sério coleciona técnicas especiais. Este capítulo apresentou algumas técnicas de codificação populares e úteis, segundo meu entendimento, claro. Existem muitas outras que ficaram de fora, sejam usadas pelas comunidades Java, Ruby, JavaScript, PHP, DotNet, Python, enfim, cada plataforma uma cultura. Existem outros livros ótimos sobre o tema, como o sensacional *Java Efetivo* de Joshua Bloch (ex-Engenheiro Chefe da Sun Microsystems) que foi inspirado no *C++ Efetivo* (outra excelente obra).

No capítulo seguinte *seguiremos o baile*, agora em como re-escrever códigos para aumentar sua qualidade, prática conhecida como **Refatoração**.

Capítulo 011 – Melhorando Códigos Existentes

Códigos que humanos possam entender...

Qualquer idiota é capaz de escrever códigos que um computador possa entender. Bons programadores escrevem códigos que seres humanos podem entender.

– Martin Fowler

O capítulo anterior abordou os problemas com código ininteligível e as técnicas para escrever um bom código. Neste capítulo continuamos trabalhando a qualidade do código, mas com o pressuposto de que é um **código legado**[42], ou seja, um código já existente escrito por ti ou por outros programadores, muitas vezes já está *"funcionando"* em *produção*[43].

> **Herança Maldita**
>
> É o nome de um *filme trash* e, informalmente, como chamamos os *códigos legados.*

Primeiro, vamos para a aceitação: nessa profissão de programador nem sempre escrevemos códigos *"do zero"*. Na verdade, frequentemente trabalhamos em cima de código alheio, ou até naquele código próprio, feito há tanto tempo que nem lembramos mais e parece de outra pessoa de qualquer maneira. A maioria dos programadores que conheço detestam trabalhar com código legado, em especial se está mal escrito ou tem uma lógica confusa, mas não há saída, temos que fazê-lo. Este é o momento de fazer uma escolha[44], *Mr. Anderson*:

1. escrever sobre esse código *"feio"* acrescentando mais código e mantendo o padrão de *feiura*;
2. rescrever, para melhorar a qualidade do código legado, antes de acrescentar código novo.

Os amadores ficam com a primeira opção.

Por desconhecimento, medo ou negligência, programadores amadores tendem a *"não tocar"* em código que funciona, mesmo que tenham a pior aparência (o código, não o programador) possível, acabando por introduzir código novo *"no mesmo sentido"* do antigo, contribuindo ainda mais para o *crescimento desordenado* do programa[45].

[42] o código se torna legado logo no primeiro minuto depois de escrito.

[43] **produção** é o nome do ambiente onde o sistema é implantado e usado pelo usuário final.

[44] problemas de escolha sempre estão nas filmografias clássicas, por exemplo em http://youtu.be/sGd8LXiGDzw.

[45] esse incremento de desordem na base de código é conhecido como Dívida Técnica e é um assunto muito discutido nas *"rodas"* de Engenharia de Software. O *site* da InfoQ Brasil tem uma coleção de artigos sobre isso: http://www.infoq.com/br/technicaldebt/.

Programadores Profissionais ficam com a segunda opção.

Os profissionais reconhecem a importância de uma base de código inteligível, por isso eles tendem a deixar os lugares por onde passam mais organizados do que quando pegaram (também conhecida como a *regra de escoteiro*).

Quem escreveu essa m*?

Não vou esconder, eu mesmo já fiz essa pergunta enquanto trabalhava em código legado. Até um dia em que me deparei com um código antigo (em Visual Basic) e horrendo, *"– quem escreveu essa m*?"* No fim do arquivo, a revelação: *"Márcio Torres"*. Ah, fui eu. Obs.: sim, eu já programei em *Visual Basic* e não é tão ruim quanto parece :)

O processo de rescrever código existente para aumentar sua qualidade é uma prática conhecida como refatoração, tema central desse capítulo e discutido nos tópicos a seguir.

Definindo Refatoração

Jogando código fora...

Um dos meus dias mais produtivos foi jogando fora 1000 linhas de código.

– Ken Thompson

A definição de refatoração é simples: é a alteração de códigos existentes sem alterar o comportamento observado do *software*. Em outras palavras, tu alteras códigos que já existentes para melhorar a qualidade interna do *software*, entretanto para o usuário final o *software* segue funcionando como antes, ou seja, a qualidade externa não é alterada.

Cuidado! Geralmente refatorar não apresenta progresso visível.

Tudo bem gastar algumas horas do dia refatorando códigos. Contudo, como refatoração não apresenta progresso visível para clientes e gestores de projeto, gastar alguns dias refatorando pode passar a imagem de que estás *"vagabundeando"*. Se é o caso e a empresa usa algum sistema de controle de tarefas (Redmine, Mingle, Trac, etc), registre tudo.

Código *ruim* que funciona

Já trabalhei em *software legado* com códigos mal escritos. Entretanto, acreditem, a maioria desses *softwares* funcionavam bem. O problema do código só aparece no momento de introduzir ou alterar funcionalidades, que exigia mais *"processamento mental"* do que o normal para entender a lógica nos códigos, sem falar dos *"hieróglifos"* deixados pelos meus queridos colegas

de profissão >:|

É importante não confundir código com baixa qualidade técnica com código que não funciona. Existem muitas classes e variáveis, muitos métodos e construções em geral, que são confusos e ineficiente, mas eficazes, isto é, eles fazem o que era suposto fazer. Por exemplo, considere o código a seguir que formata um número qualquer para 9 dígitos completando com zeros à esquerda:

Da série: códigos que te fazem chorar

```
<?php
// formatando o número para 9 dígitos (da maneira mais ingênua possível)
if ($numero_nota >= 1 && $numero_nota < 10) {
    $numero_nota = '00000000' . $numero_nota;
} else if ($numero_nota >= 10 && $numero_nota < 100) {
    $numero_nota = '0000000' . $numero_nota;
} else if ($numero_nota >= 100 && $numero_nota < 1000) {
    $numero_nota = '000000' . $numero_nota;
} else if ($numero_nota >= 1000 && $numero_nota < 10000) {
    $numero_nota = '00000' . $numero_nota;
} else if ($numero_nota >= 10000 && $numero_nota < 100000) {
    $numero_nota = '0000' . $numero_nota;
} else if ($numero_nota >= 100000 && $numero_nota < 1000000) {
    $numero_nota = '000' . $numero_nota;
} else if ($numero_nota >= 1000000 && $numero_nota < 10000000) {
    $numero_nota = '00' . $numero_nota;
} else if ($numero_nota >= 10000000 && $numero_nota < 100000000) {
    $numero_nota = '0' . $numero_nota;
} else if ($numero_nota >= 100000000 && $numero_nota < 1000000000) {
    $numero_nota = '' . $numero_nota;
}
```

https://gist.github.com/marciojrtorres/c9c8d80e26c04cbc94ff

Do ponto de vista do usuário final, esse código é perfeito, pois ele cumpre a especificação. Quer dizer que na tela do usuário final um número será apresentado com até 8 zeros a esquerda. Programadores espertos acham esse código bobo e deixam ele pra lá – afinal, ele funciona. Programadores profissionais reescrevem este código. A propósito, podes usar este código como um exercício, rescrevendo (isto é: **refatorando**). Se esse foi fácil de entender, aqui vai outro (copiei e colei exatamente como encontrei):

Esse também é de gosto dúvidoso

```
<?php
$colunaDiferente = $_SESSION['janelaBusca'][$_POST['idJanela']]['colunaDiferente'\
];

if(is_array($colunaDiferente)){
    if (is_array($vet)){
        foreach ($vet as $k=>$v){
           foreach ($v as $k1=>$v1){
                foreach ($v1 as $k2=>$v2){
                    foreach ($colunaDiferente as $k3=>$v3){
                        if ($vet[$k]['r'][$k2]['c'] == $k3) {
                            $vet[$k]['r'][$k2]['c'] = $v3;
                        }
                    }
                }
            }
        }
    }
}
```

https://gist.github.com/marciojrtorres/60f8e6c686852070b139

Geralmente, não precisa examinar muito para perceber que um código precisa de refatorações, no entanto, o julgamento do que precisa ser refatorado é bem subjetivo, pois a noção de qualidade é subjetiva. Quero dizer: dar um problema para 10 programadores diferentes resultará em 10 soluções funcionais, mas diferentes. Por isso, as refatorações são documentadas e organizadas conforme os *"padrões de deficiência"* encontrados nos códigos, conhecidos como ***maus cheiros no código*** ou *Code Smell* (originalmente). Os principais tipos de *smells* serão listados no decorrer desse capítulo.

Por que refatorar?

A dúvida comum dos programadores amadores é: se refatorar significa rescrever um código que já funciona, então, "*se ele funciona por que tenho de refatorar?*". Também conhecido como *"se funciona tá 'serto'"(sic)*[46].

Posso te apresentar vários argumentos, pois existem muitos benefícios em refatorar. Por exemplo, considere as motivações a seguir:

- **Refatorar melhora o projeto do software**: o projeto do *software* costuma deteriorar, agravando com ciclos frequentes de manutenção. O código tende a ficar desorganizado e a refatoração ajuda a eliminar código sem utilidade, repetido e ofuscado.

[46] *"Se funciona tá serto!"* (sic) – Princípio Fundamental da Programação Orientada a Gambiarras (PPOG): http://goo.gl/G9GZjJ.

- **Refatorar torna o software mais fácil de entender**: a refatoração elucida e ilumina o código, tornando-o compreensível não só para ti, mas para a equipe inteira (repita comigo: programar não é uma atividade solitária). A refatoração é feita pensando em todos que um dia terão de ler o código. A inteligibilidade do código diminui o esforço mental para entendê-lo e permite que tua equipe seja mais produtiva.[47]
- **Refatorar ajuda a encontrar falhas**: alguns *bugs* são ainda mais difíceis de encontrar em códigos bagunçados. A refatoração pode trazer outro ponto-de-vista ao código, ajudando a revelar erros que antes não eram tão óbvios.
- **Refatorar ajuda a programar mais rapidamente**: um código confuso vai sempre te tomar mais tempo para ser compreendido e alterado. Códigos inteligíveis tendem a reduzir o esforço para compreensão e te permitem realizar alterações com mais segurança e em menos tempo.
- **Refatorar ajuda a reduzir o risco de doenças cardiovasculares**: segundo pesquisas, programadores que refatoram seu código tem menor tendência a infartar (calma, não é sério :P).

> **"Corte o fio vermelho..."**
>
> Sabe aqueles filmes onde existe um dispositivo de destruição em massa e o herói deve cortar um fio colorido para desarmá-lo? Mas, claro, ele está sob uma luz neon que não permite distinguir a cor dos fios. Então, penso que o medo de alterar um código confuso e quebrar a p**** toda te dá uma sensação bem semelhante a esta.

Maus cheiros no código

Sabendo o que é refatorar e porque refatorar, agora vem o *quando refatorar?*. Uma heurística usada para responder essa pergunta é a detecção de *"maus cheiros"* no código (é isso mesmo), também conhecido simplesmente por *smell*[48] (como será mencionado no restante desse capítulo). É uma analogia, segundo qual ao ler o código é possível identificar partes *"estranhas"* que, digamos, *cheiram!*

Alguns códigos têm *smells* que se enquadram desde o (simplesmente) ambíguo até o que pode chamar-se de incompreensível. Os *smells* foram documentados por vários profissionais renomados, especialmente Martin Fowler – a maior parte de *smells* e refatorações presentes nesse capítulo são inspirados nas suas obras.

Nos tópicos a seguir serão enumerados os *smells* mais comuns. Não é uma exaustão do tema, muito pelo contrário, daria um livro só disso. Entretanto, serve como um *"caminho da pedras"* para todo programador iniciante que espera tornar-se um profissional digno e valioso.

[47] se tu és um Gerente de Projetos (ou aspira a ser um), pense em usar aqueles dias em que sua equipe não está alocada em projetos novos para refatorar os projetos antigos.

[48] O termo *Code Smell* foi cunhado por Kent Beck e é usado em todas as referências sobre refatoração, incluindo a notável obra de Martin Fowler: Refatoração. Mais sobre o termo pode ser visto aqui [EN] http://martinfowler.com/bliki/CodeSmell.html e aqui [EN] http://c2.com/cgi/wiki?CodeSmell.

Smell: código duplicado

Este é, penso, o *smell* mais frequente nos códigos por aí. É relativamente fácil identificar código duplicado, os fragmentos não precisam ser exatamente idênticos, mas sempre vai te dar aquela sensação de que alguém fez um `CTRL+C CTRL+V` e mudou uma ou duas partes.

Código duplicado é péssimo para o projeto, pois alterações neste código têm de ser feitas em todos os lugares que ele aparece (onde foi *"colado"*). Entre as refatorações típicas para código duplicado destacam-se a introdução de novas classes e métodos para representar, de uma só vez, este código e permitir sua reutilização.

Copiar e colar TUDO é uma prática amadora.

Arrastar, copiar, desfazer, etc, são facilidades que as interfaces de usuários introduziram. Contudo, parece haver um *"vício"* por certas funcionalidades, como o COPIAR/COLAR. É uma prática aplicada (e útil) logo no aprendizado, vejo muito isso nos cursos, mas se não for controlado acontecem situações bizarras, por exemplo já vi copiarem uma classe inteira só para aproveitar o `package;`, `public class` e chaves, realizando mais esforço para esvaziar o *"miolo"* do que se tivessem escrito a classe *"do zero"*.

Smell: classes e método muito longos

Não existe uma métrica padrão para o tamanho de uma classe e/ou de um método, mas saiba que quanto maior uma classe ou método há mais chances deles terem de mudar, além da dificuldade de saber onde mudar. Tente, sempre que possível, quebrar métodos longos em métodos menores e classes muito longas em mais e menores classes. Lembre sempre que quanto menor, mais fácil de dar manutenção.

Programadores Profissionais realmente não se importam em criar várias classes pequenas.

Enquanto programadores amadores relutam em criar mais classes os profissionais sempre criam uma quando veem uma oportunidade. Classes pequenas e até *tiny*[49] são mais fáceis de ler, entender e alterar.

Smell: excesso de parâmetros

A quantidade excessiva de parâmetros já foi discutida no Capítulo 1 em Poucos parâmetros, menos é mais nesse caso. Métodos com muitos parâmetros dificultam o entendimento e aumentam o acoplamento. Existem algumas formas de reduzir a quantidade de parâmetros como a refatoração Introduzir Objeto Parâmetro que será vista adiante neste capítulo.

[49] *Tiny Types* são classes minúsculas, geralmente imutáveis. Se quiser saber mais, no blog da Caelum tem um artigo muito bom sobre o assunto: http://blog.caelum.com.br/pequenos-objetos-imutaveis-e-tiny-types/.

Smell: grupos de dados

Os dados que andam juntos devem estar contidos em sua própria classe e ter seus próprios objetos. O *smell* Grupo de Dados é visível em algumas classes quando alguns atributos parecem estar fora do contexto, como se fossem de outro objeto. Esses grupos são, geralmente, extraídos (recortados) e introduzidos (colados) em uma nova classe. Um amostra desse *smell* pode ser vista no código a seguir:

Classe com smell Grupo de Dados

```
package comgrupo;


public class Funcionario {
  int    codigo;
  String nome;
  String razaoSocial;
  String cnpj;
  String rua;
  int    numero;
  String bairro;
  String cep;
  String telefone;
  String inscricaoEstadual;

}
```

No exemplo anterior, a classe `Funcionario` contém dados do funcionário, porém deixa espaço para uma má interpretação do modelo, devido ao *grupo de dados* que se formou. Considere o atributo `numero`, embora ele represente o número do endereço ainda assim pode ser mal interpretado durante a utilização, por exemplo:

```
// é o número do funcionário?
fieldNumero.setText(funcionario.numero);
// na verdade é o número do endereço do funcionário!
```

Com um pouco de observação se percebe que `rua`, `numero`, `bairro` e `cep` andam juntos porque representam um *Endereço* (que deve ser um objeto separado). Esse dados devem estar em uma classe que dê significado à eles. Embora a solução seja simples, nós vemos alguns *"contornos"*, como a adição de comentários:

Comentar – infelizmente um dos primeiros recursos

```
package comgrupocomentado;


public class Funcionario {
  int    codigo;
  String nome;
  String razaoSocial;
  String cnpj;
  // endereço do funcionário
  String rua;
  int    numero;
  String bairro;
  String cep;
  // fim endereço
  String telefone;
  String inscricaoEstadual;

}
```

Outra opção, também deselegante, para esclarecer os atributos é usar a *notação húngara*, fazendo uso de prefixos[50]:

Prefixar é, dependendo, uma solução deselegante

```
package comgrupoprefixado;


public class Funcionario {
  int    codigo;
  String nome;
  String razaoSocial;
  String cnpj;
  String enderecoRua; // prefixo endereco
  int    enderecoNumero;
  String enderecoBairro;
  String enderecoCep;
  String telefone;
  String inscricaoEstadual;

}
```

Comentários e prefixos são apenas formas de adiar o problema e caracterizam uma resistência à criação de classes conhecida como Obsessão Primitiva que será visto a seguir.

Para encerrar esse tópico a seguir está a solução adequada (extrair classe):

[50] prefixar campos é uma prática discutível (alguns acham boa e outro não) que é conhecida como *Notação Húngara*, servindo para identificar variáveis, como `int intQuantidade = 12` ou `TextBox tbNome = new TextBox()`.

Fazendo o certo: separar Funcionario e Endereco

```
package semgrupo;

public class Funcionario {
  int    codigo;
  String nome;
  String razaoSocial;
  String cnpj;
  String inscricaoEstadual;
  String telefone;

  Endereco endereco = new Endereco();
}
```

Acessando o número do endereço do funcionário

```
    Funcionario f = new Funcionario();
    f.endereco.numero = 122;
  }
```

A separação de dados pode melhorar a semântica do projeto e também promover a modularidade e a reutilização, além de promover classes pequenas como (ver Classes e métodos muito longos).

Smell: obsessão primitiva

Os tipos de dados podem ser divididos em duas categorias: tipos primitivos/básicos e complexos. Os *primitivos* são uma simples representação binária de seu valor, que ocupam pouca memória se comparado aos complexos. Por exemplo, o tipo inteiro da linguagem Java (`int`) utiliza 32 bits (4 bytes) para armazenar um número. Os tipos *complexos* são classes ou estruturas com dados e lógica, ocupam mais memória e têm um ciclo de vida diferente dos tipos primitivos (na maioria das linguagens), como exemplo o tipo `Integer`, que utiliza os 32 bits do `int` e mais uma quantidade extra de bits para prover a infraestrutura do objeto.

Uma decisão importante do desenvolvedor é quando usar um tipo primitivo ou complexo para representar uma informação. A obsessão primitiva é observada quando a maior parte das informações são representadas com tipos primitivos, mesmo onde eles não são adequados.

Fazer tudo com *strings* também é considerado uma *Obsessão Prmitiva*

Na linguagem Java (e outras), *strings* são tipos complexos (objetos), mas participam do *smell Obsessão Primitiva*. Codificar dados como *strings*, onde um tipo específico seria mais apropriado, é um *smell* bem frequente. Não é incomum programadores amadores usarem uma *string* para guardar listas de palavras (separadas por vírgula) quando um vetor, por exemplo, seria mais apropriado.

A refatoração usada para o *smell* Obsessão Primitiva é substituir o primitivo (básico) por um complexo, ou seja, em vez de usar `int`, `double`, `String`, etc, usar um tipo complexo (uma classe/um objeto) mais adequado. Para entender melhor considere o código a seguir:

Um exemplo do smell obsessão primitiva no campo peso

```
class Produto {
  int    codigo;
  String descricao;
  int    peso;
}
```

O foco de discussão é o atributo `peso`, representado como um `int`. Representar *peso* como inteiro parece bem óbvio, mas a questão é que um *peso*, enquanto medida, é representando por um número e uma unidade. Então vem a primeira dúvida: **`peso` do `Produto` está em qual unidade de medida?** Entre vários modos de resolver esse impasse, a solução mais ingênua seria adicionar um comentário, como segue:

Solução ingênua para dar significado a um primitivo: adicionar um comentário

```
class Produto {
  int    codigo;
  String descricao;
  int    peso; // em gramas
}
```

É uma *"solução ingênua"* porque não passa de um *placebo*. Por exemplo, alguém que utilize a classe `Produto` ao acessar o `peso` ainda não saberia se é em gramas, já que não está lendo o código-fonte da classe `Produto`, por exemplo: `produto.peso = 100`. Ver código a seguir:

Situação de uso de um atributo primitivo

```
Produto produto = new Produto(); // até aqui tudo bem
produto.peso = 100; // 100 gramas, libras, kilos ou toneladas?
```

O código anterior esclarece melhor o problema. Nao há como saber a unidade sem ler o código-fonte (que nem sempre está disponível, poderia ser uma referência à unidade compilada de `Produto`). Uma solução *"menos pior"* seria sufixar, como no exemplo a seguir:

Sufixar/Prefixar: solução paliativa

```
class Produto {
  int    codigo;
  String descricao;
  int    pesoEmGramas;
}
// situação de uso:
Produto produto = new Produto();
produto.pesoEmGramas = 100; // um pouco melhor
```

Este último código ficou melhor, contudo é uma solução temporária no tempo de vida de um *software*. Por exemplo, pode mudar o requisito (e eles sempre mudam) para que o *peso* possa ser informado em várias unidades de medida como kilos, gramas ou libras. A solução mais básica, a nível de código (mas péssima a nível de projeto), é adicionar uma variável de tipo, também primitiva/básica, como no exemplo a seguir:

A obsessão primitiva aumenta se for adicionada uma variável de tipo

```
class Produto {
  int    codigo;
  String descricao;
  int    tipoPeso; // 0: grama, 1:kilo, 3:libras
  int    peso;
}
// situação de uso:
Produto p = new Produto(); // até aqui ok
p.tipoPeso = 1; // hã?
p.peso = 200;
// sem conhecer o código-fonte de Produto, quanto pesa?
```

Neste ponto já é possível notar claramente o problema da *Obsessão Primitiva*, caracterizada por, insistentemente, modelar atributos com `int`'s, `double`'s, `String`'s, etc. Na verdade, representar uma informação com dois atributos (`tipoPeso` e `peso`) é também um Grupo de Dados (ver tópico anterior). Programar de forma Orientada a Objetos implica em, claramente, escrever classes e construir objetos. A solução adequada é introduzir uma classe (que são meios de representar tipos complexos), semelhante a classe `Peso` que está no código a seguir:

Introduzindo novas classes: um remédio para a obsessão primitiva

```
class Peso {
  int gramas;

  Peso(int gramas) { this.gramas = gramas; }
  static Peso emKilos(int k) { return new Peso(k * 1000); }
  int kilos() { return this.gramas / 1000; }
}

class Produto {
  int    codigo;
  String descricao;
  Peso   peso;
}

// situação de uso:
Produto produto = new Produto();
produto.peso = Peso.emKilos(2);
```

Partir para a representação adequada com a classe `Peso` possibilita encapsular o atributo e disponibilizar os métodos necessários para operar as diferentes representações. O exemplo ilustra a situação de *kilos*, facilitado com um Método Fábrica Estático, visto no Capítulo 10, mas pode ser usada similarmente para libras, toneladas, etc.

Resistir em criar classes e tipos novos é uma atitude amadora.

Programar envolve projetar, além de só escrever códigos. Projetar significa pensar mais amplamente, levando geralmente a separação do programa em módulos. Em POO esses módulos são classes, sem falar dos diversos princípios do paradigma orientado a objetos. Quando não-POO, pelo menos se fala em arquivos, componentes, sub-rotinas, enfim, fugir do modelo monolítico com os recursos disponíveis.

Smell: comentários (também conhecidos como *"desodorante"*)

As noções da utilizade de comentários já foram discutidas no tópico Não introduzirás comentários inúteis. O problema de muitos comentários é que eles podem estar ali para explicar um código difícil de entender devido a sua baixa qualidade. Em outras palavras, o código não tem um *"bom cheiro"* e os comentários estão ali para suprimir o cheiro. Este tipo decomentário é conhecido, por isso, como um *smell* chamado *"desodorante"*. Claro que, frequentemente, comentários são melhores do que nada, mas lembre sempre do mantra: *"Bom código é sua melhor documentação..."*.

Smell: números (e strings) mágicos

Números, *strings* e outros literais soltos no código é um dos *smells* mais comuns e fáceis de identificar. O algoritmo, as instruções, dependem de dados dinâmicos e estáticos. Os dinâmicos são as variáveis e os estáticos são chamado de constantes. Existem constantes nomeadas, como `final double PI = 3.1415`, e literais, isto é, só o número (solto). Sempre que uma instrução usa um literal que não tem um significado claro e evidente, chamamos esses literais de números mágicos (ou *strings* mágicas). A seguir um pequeno exemplo:

Qual o significado de status 2?

```
public void imprime(Documento doc) {
  if (doc.getStatus() != 2) return;
  if (doc.getMargemDireita() > 400)
    doc.setMargemDireita(380);
// ...
```

No código anterior existem três constantes literais: `2`, `400` e `380`. Enquanto a margem podem ser inferida, o `status` não é muito claro – o que significa `status != 2`? Números e outros literais no código dificultam o entendimento e devem ser substituídos, sempre que possível, por constantes nomeadas. Na prática, não é muito complicado de se fazer, todas as linguagens oferecem construtos para declarar e definir constantes. Considere o exemplo a seguir:

Dê nome aos literais, use constantes, não irás te arrepender

```
// Em Java as constantes são
// declaradas com: static final tipo nome = valor
static final int NOVO = 0;
static final int ALTERADO = 1;
static final int SALVO = 2;

public void imprime(Documento doc) {
  if (doc.getStatus() != SALVO) return;
// ...
```

Mais adiante, a introdução de constantes será melhor discutida. Antes de fechar este tópico, é importante saber que condicionais confusos podem ser substituídos por um método consulta, como no código a seguir:

Qual o significado de status 2?

```
// método consulta
public boolean naoEstaSalvo(Documento doc) {
  return doc.getStatus() != 2;
}

public void imprime(Documento doc) {
  if (naoEstaSalvo(doc)) return;
  if (doc.getMargemDireita() > 400)
    doc.setMargemDireita(380);
// ...
```

O Método consulta também será discutido mais a frente neste capítulo.

> **Dá-le bruxo, a URL do banco *tá hard coded*... (sic) – *Hardioquê?!***
>
> Isso mesmo, *hard coded*. Informações *hard coded* são aquelas escritas literalmente (fixas) no código. Por exemplo: `String db = "jdbc:mysql://10.122.33.2:3306/progpro` é uma informação *hard coded*. *"Hardcodar"* é uma má prática. Mesmo introduzindo uma constante, certos dados não deveriam ser *"hardcodados"*, a instrução de conexão com uma base de dados é um bom exemplo, pois para fazer mudanças de servidor, senha, etc, seria necessário alterar os fontes. Em outras palavras, *"hardcodar"* dados aumenta a probabilidade de ter que alterar o código – o que é ruim, só para lembrar. Neste caso em particular, poderia ser substituído por um arquivo externo de configuração ou variável de ambiente, por exemplo: `String db = System.getProperty("db")`.

Refatorações Comuns

Existem várias refatorações para remover os *smells* apresentados nos tópicos anteriores, algumas simples, como *renomear*, e outras mais complexas, como *extrair/introduzir superclasses*. Nos tópicos a seguir serão apresentadas algumas refatorações típicas usadas para remover os *smells* que ocorrem com mais frequência.

Refatorações e a Programação Orientada a Objetos

Algumas refatorações estão associadas a *smells* de POO, como problemas de acoplamento, nível de abstração dos atributos e métodos, etc. Não é objetivo desse livro tratar de POO, então é importante que tenhas uma noção, pelo menos, dos princípios básicos de POO. Vai te ajudar a entender o porquê de certas refatorações.

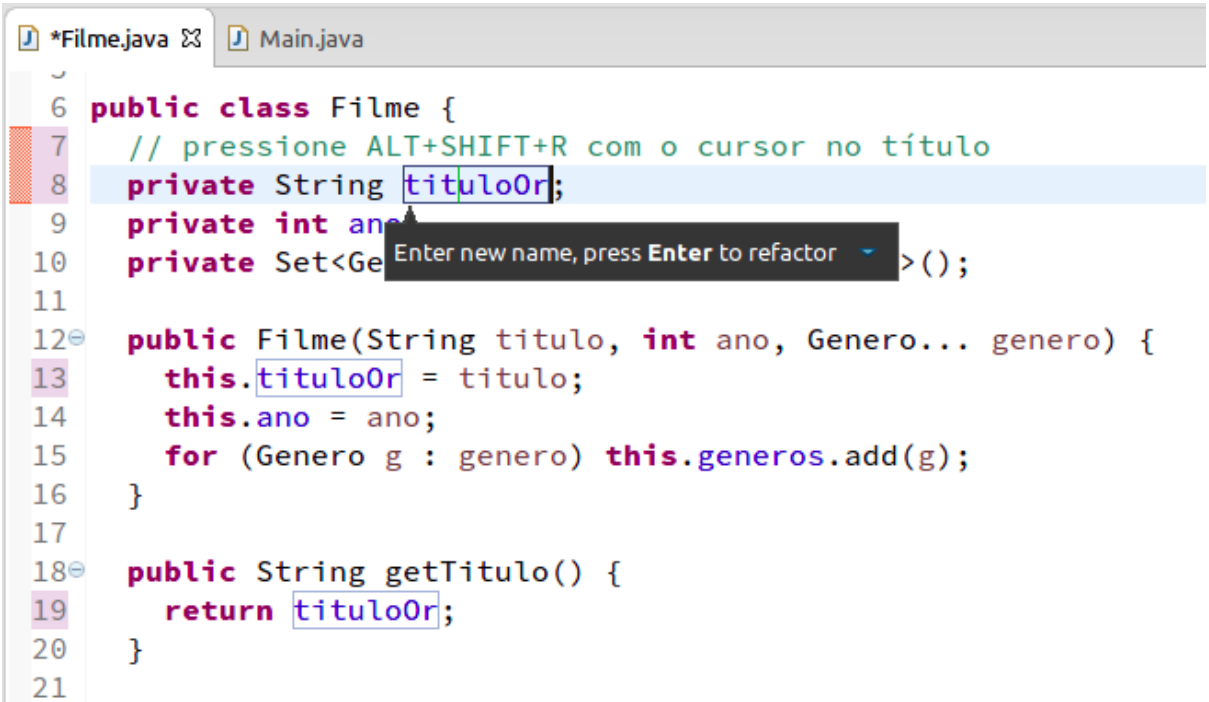
Refatoração: renomear

Alterar o nome de classes, métodos e variáveis é a refatoração mais comum, simples e que traz muitos benefícios. Ao mesmo tempo, *renomear* é uma refatoração temida pelos programadores, pois alterar o nome de estruturas já desenvolvidas incorre, às vezes, numa *"quebradeira"* no código, devido geralmente as referências espalhadas pela base de código. Em outras palavras, renomear uma classe implica em referenciá-la pelo novo nome em todos os locais em que ela é usada, bem como se mudares o nome de uma propriedade pode fazer com que um campo suma da interface com o usuário, se também não for atualizada.

O primeiro nome que vem a cabeça quase sempre não é o melhor nome possível

Os programadores devem trabalhar psicologicamente com o fato de que os nomes dados a variáveis, classes, métodos, etc, são temporários. Por mais que se procure o melhor nome, a opinião pode mudar no dia seguinte. Por isso é importante usar o *renomear* sem medo.

O ato de *renomear* é um trabalho árduo (e chato) se feito manualmente, mas as IDE's possibilitam a renomeação assistida de variáveis, classes e seus membros (atributos e métodos), onde o assistente será responsável por atualizar as referências em todas as ocorrências do elemento renomeado. Por exemplo, para renomear uma variável é possível pressionar `CTRL+R` no NetBeans ou `ALT+SHIFT+R` no Eclipse. Considere o trecho de código a seguir:



Renomear em ação no Eclipse

Considere que queiramos alterar o nome da variável `titulo` para `tituloOriginal`. A imagem demonstra o que acontece ao pressionar `ALT+SHIFT+R` na IDE Eclipse. Perceba que todas as

ocorrências são alteradas. Ainda, o método `getTitulo()` também deve ser renomeado para `getTituloOriginal()`, bastando para isso pressionar novamente `ALT+SHIFT+R` agora com o *cursor* sobre o método, ou usar o menu: *Refactor -> Rename*.

Renomeando sem uma IDE

Se tu não estás escondido num *bunker* e isolado do mundo, deves ter percebido que há um movimento favorável a editores leves, nem tão *"pelados"* nem tão *"sobrecarreados"*, como o Sublime Text, Atom, Visual Studio Code, etc. Os 3 citados oferecem um atalho (`CTRL+D`) para selecionar vários trechos e editá-los simultaneamente. Contudo, para editar código que tenha referência em outros arquivos, usar um editor comum vai te trazer mais trabalho, mas não é impossível. Engenharia de Software não é Engenharia Civil, código não é concretado, então pode (e deve) mudar para aumentar a qualidade sempre que houver oportunidade.

Refatoração: extrair (introduzir) classe

Extrair Classe é uma refatoração bastante comum para encapsular Grupos de Dados ou pelo menos mover responsabilidades de *uma classe que faz coisas demais*[51] para outras classes menores. A mecânica dessa refatoração é bastante simples: criar uma nova classe, mover os atributos e métodos para ela e por fim referenciá-la no lugar onde estavam os dados. Veja a seguir um exemplo simples:

Classe Cliente com o Grupo de Dados telefone

```
class Cliente {
  Integer codigo;
  String  nome;
  // telefone
  String  ddd;
  String  numero;
  // fim telefone
  String  CPF;
```

Observando o exemplo, a classe `Cliente` apresenta dois atributos (`ddd` e `numero`) que deveriam ser de responsabilidade de outra classe. Na prática, têm-se os mesmos dados, apenas em classes diferentes e representados de outra forma. Para refatorar pode-se criar uma classe `Telefone` e mover os atributos `ddd` e `mumero` para ela. Em seguida substitui-se `ddd` e `numero` em `Cliente` por `Telefone telefone = new Telefone()`, conforme a versão refatorada a seguir:

[51] Classes que exercem muitas funcionalidades e têm muitas responsabilidades também é um *smell* chamado *God Class*.

Classe Cliente refatorada

```
class Cliente {
  Integer codigo;
  String  nome;
  Telefone telefone = new Telefone();
  String  CPF;
```

A extração (introdução) de uma classe traz vantagens como: reuso, coesão, encapsulamento e melhor nível de abstração para projeto (princípios de POO). O efeito colateral é o aumento do grafo de objetos e de indireção (um objeto com referência a outro, que por sua vez faz referência a outros e assim por diante). Para acessar certas propriedades é necessário navegar no grafo, como no exemplo a seguir:

Usando a classe Clientes antes e depois de ser refatorada

```
// antes de extrair o Telefone:
Cliente cliente = new Cliente();
cliente.codigo = 1;
cliente.nome = "Um Nome";
cliente.ddd = "99";
cliente.numero = "44556677";

// depois de refatorar:
Cliente cliente = new Cliente();
cliente.codigo = 1;
cliente.nome = "Um Nome";
// ddd do Telefone do Cliente
cliente.telefone.ddd = "99";
cliente.telefone.numero = "44556677";
```

Quando estiver em dúvida se deve ou não aumentar a indireção, pense que programas monolíticos são úteis para criar um utilitário ou pequena aplicação. Qualquer coisa que vai além daqueles *"programinhas"* didáticos vai precisar modularizar o suficiente para minimizar os pontos de mudança.

> **Não aguento mais criar classes :'(...**
>
> É uma típica reclamação dos estudantes nas disciplinas de Programação Orientada a Objetos. Para motivar, pense que construir sistemas é como um tipo de engenharia. Com materiais elementares se constrói pequenos blocos e desses blocos se constrói módulos e desses módulos se constrói módulos maiores ainda até se chegar no produto final.

Refatoração: extrair (introduzir) superclasse

Essa refatoração é bem semelhante a Extrair Classe, mas com uma diferença: o *Extrair Superclasse* busca mover membros (atributos e métodos) para uma nova classe que será usada como superclasse da classe refatorada. A mecânica consiste em criar uma nova classe, mover os atributos e métodos para ela, e por fim estendê-la. A seguir um exemplo:

Extraindo atributos comuns para uma superclasse

```
// classe original a ser refatorada:
class Cliente {
  Integer codigo;
  String  nome;
  String  CPF;
  String  razaoSocial;
  String  CNPJ;
  Date    dataCadastro;
// ...

// pode ser restruturada assim:
abstract class Cliente { // atributos comuns
  Integer codigo;
  Date    dataCadastro;
// ...
// atributos específicos:
class ClientePessoaFisica extends Cliente {
  String  nome;
  String  CPF;
// ...
// atributos específicos:
class ClientePessoaJuridica extends Cliente {
  String  razaoSocial;
  String  CNPJ;
// ...
// situação de uso:
ClientePessoaFisica cliente = new ClientePessoaFisica();
cliente.codigo = 3; // é acessível por herança
```

Na classe `Cliente` alguns atributos são opcionais, conforme o **tipo** de cliente, tornando-a uma boa candidata à essa refatoração.

Extrair Classe vs Extrair Superclasse == Refatorar para Composição vs Refatorar para Herança

Se tens bons conhecimentos de POO deves conhecer as diferenças entre herança, agregação e composição. As refatorações de extração de classe e superclasse têm tudo haver com esses princípios fundamentais de orientação a objetos. Enquanto extrair uma classe e usá-la como atributo é uma **composição** extrair e usá-la como superclasse é uma **herança**. A aplicação é caso-a-caso, não poderíamos, no exemplo anterior, fazer `Cliente` estender `Telefone`, não faz nenhum sentido, é só lembrar do *"é um"* e *"tem um"*, como *Cliente tem um Telefone* e não *Cliente é um Telefone*.

Refatoração: extrair (introduzir) método

É outra refatoração típica e prática, usada para remover código duplicado ou esclarecer uma lógica confusa. Em caso de código duplicado, ele pode ser extraído (daí o nome) para um novo método e reutilizado sempre que necessário. A mecânica consiste em implementar um novo método com um nome claro e objetivo e então mover o código para este novo método, satisfazendo as dependências nos parâmetros do método.

Os IDE's disponibilizam assistentes que executam esta mecânica; no NetBeans e Eclipse, procure no menu refatorar (*Refactor*) a opção *"Extrair (ou Introduzir) método"*. No NetBeans, selecione as instruções e pressione `SHIFT+ALT+M`. A seguir um exemplo:

Lógica compartilhada entre os métodos: smell candidato a extração

```
List<Ator> buscaTodos() {
  List<Ator> atores = new ArrayList<Ator>();
  Connection conexao = ConnectionFactory.newConnection();
  try {
    String SQL = "SELECT * FROM atores";
    PreparedStatement comando = conexao.prepareStatement(SQL);
    ResultSet resultado = comando.executeQuery();
    while (resultado.next()) {
      Ator ator = new Ator();
      ator.setId(resultado.getInt(1));
      ator.setNome(resultado.getString(2));
      ator.setSobrenome(resultado.getString(3));
      ator.setVersao(resultado.getTimestamp(4));
      atores.add(ator);
    }
// ...
List<Ator> buscaPorNome(String nome) {
  List<Ator> atores = new ArrayList<Ator>();
  Connection conexao = ConnectionFactory.newConnection();
  try {
    String SQL = "SELECT * FROM atores WHERE nome = ?";
    PreparedStatement comando = conexao.prepareStatement(SQL);
```

```
23      comando.setString(1, nome);
24      ResultSet resultado = comando.executeQuery();
25      while (resultado.next()) {
26        Ator ator = new Ator();
27        ator.setId(resultado.getInt(1));
28        ator.setNome(resultado.getString(2));
29        ator.setSobrenome(resultado.getString(3));
30        ator.setVersao(resultado.getTimestamp(4));
31        atores.add(ator);
32      }
33  // ...
```

Os métodos `buscaTodos` e `buscaPorNome` compartilham a lógica de mapeamento e população da lista de atores, considere o código entre as linhas 8-15 e 25-32, eles são idênticos. Esta lógica pode ser extraída para um novo método que será executado em seu lugar. Este novo método tem que receber as dependências necessárias para concluir seu trabalho, que neste exemplo são as variáveis: `resultado` e `atores`. Para comparar, a seguir está a versão refatorada:

Lógica compartilhada extraída para um novo método

```
1   List<Ator> buscaTodos() {
2     List<Ator> atores = new ArrayList<Ator>();
3     Connection conexao = ConnectionFactory.newConnection();
4     try {
5       String SQL = "SELECT * FROM atores";
6       PreparedStatement comando = conexao.prepareStatement(SQL);
7       ResultSet resultado = comando.executeQuery();
8       processaResultSet(resultado, atores);
9   // ...
10  List<Ator> buscaPorNome(String nome) {
11    List<Ator> atores = new ArrayList<Ator>();
12    Connection conexao = ConnectionFactory.newConnection();
13    try {
14      String SQL = "SELECT * FROM atores WHERE nome = ?";
15      PreparedStatement comando = conexao.prepareStatement(SQL);
16      comando.setString(1, nome);
17      ResultSet resultado = comando.executeQuery();
18      processaResultSet(resultado, atores);
19  // ...
20  void processaResultSet(ResultSet resultado, List<Ator> lista) {
21    while (resultado.next()) {
22      Ator ator = new Ator();
23      ator.setId(resultado.getInt(1));
24      ator.setNome(resultado.getString(2));
25      ator.setSobrenome(resultado.getString(3));
26      ator.setVersao(resultado.getTimestamp(4));
27      atores.add(ator);
```

```
28    }
29  }
```

No código anterior, nas linhas 8 e 18, estão as chamadas para o método `processaResultSet(ResultSet, List<Ator>)`. Este método concentrou a lógica redundante. É importante comentar que **a refatoração melhora o projeto**. Antes da refatoração, se fosse necessário adicionar um campo a mais na classe `Ator`, por exemplo `biografia`, teríamos de alterar todos os métodos para mapear `ator.setBiografia(resultado.getString(5))`. Ter um método dedicado, após a refatoração, permite isolar as mudanças.

Refatoração: introduzir variável explicativa

As Variáveis Explicativas são úteis para esclarecer uma expressão condicional confusa ou mesmo uma expressão muito longa no código. Elas devem ser introduzidas logo acima das expressões que devem *explicar*. A mecânica é simples: crie uma variável *booleana* antes da expressão condicional, atribua a ela o resultado da expressão e após substitua a expressão pela variável. A seguir um exemplo:

Condicional confusa: candidata a uma variável explicativa

```
public void imprime(Documento doc) {
  if (doc.getHeader().getText().trim().length() > 0) {
// ...
```

O objetivo é tornar desnecessário o uso de raciocínio para entender a expressão condicional. O leitor deve passar os olhos e logo entender o que ela faz: verificar *se tem um cabeçalho*. A seguir o mesmo código usando uma variável explicativa:

Variável explicativa para explicar condicional confusa

```
public void imprime(Documento doc) {

  boolean temCabecalho = doc.getHeader().getText().trim().length() > 0;

  if (temCabecalho) {

// ...
```

O objetivo é adicionar clareza permitindo a leitura do código, isto é, passar os olhos por `if (temCabecalho)` é melhor que por `if (doc.getHeader().getText().trim().length() > 0)`, correto?

Refatoração: introduzir método consulta

Os métodos consulta servem como uma alternativa às Variáveis Explicativas sendo usados, também, para esclarecer uma expressão condicional. A vantagem de um método é que ele pode

ser reaproveitado, reusado em várias partes da classe. A mecânica é semelhante: crie um método com retorno *booleano*, mova para ele a lógica da expressão condicional, ajuste a assinatura do método para satisfazer as dependências da expressão e então substitua a expressão por uma chamada ao método consulta. A seguir um exemplo para esclarecer a ideia:

Refatorando condicional para um método consulta

```
// mesmo exemplo usado em Variável Explicativa
public void imprime(Documento doc) {
  if (doc.getHeader().getText().trim().length() > 0) {
// ...

// solução diferente: Método Consulta
public boolean temCabecalho(Documento doc) {
  return doc.getHeader().getText().trim().length() > 0;
}
public void imprime(Documento doc) {
  if (temCabecalho(doc)) { // chamada do Método Consulta
// ...
```

Método Consulta é uma especialização da refatoração Extrair Método, na prática, muitos problemas de código duplicado e lógica confusa podem ser solucionados com extrações (ou introduções) de novos métodos e classes.

Refatoração: inverter condicional

Os desvios condicionais ramificam o programa e adicionam complexidade (quantidade de caminhos executáveis possíveis). Tornar os desvios legiveis é importante e te fará gastar menos tempo quando voltar no código. Entre as refatorações comuns, é sugerido inverter o condicional quando a expressão faz uma negação, tornando-a uma afirmação. Na prática, a intenção é remover o operador de negação (em Java é a `!`, mas em outras linguagens pode ser o construto `not`, como em Ruby) do condicional, que pode passar despercebido por olhos cansados (naquele feriado, entrega atrasada, ...) e pouco atentos. A mecânica não é complicada, basta remover a negação e inverter as cláusulas, como no exemplo a seguir:

Invertendo um condicional negativo

```
public void imprime(Documento doc) {
  if (!doc.isCarta()) {
    prepareA4Paper();
  } else {
    prepareCartaPaper();
  }
// ...
// é mais claro verificar se o doc é carta
public void imprime(Documento doc) {
  // se é carta, prepara papel carta, OK
```

```
  if (doc.isCarta()) {
    prepareCartaPaper();
  } else {
    prepareA4Paper();
  }
// ...
```

Outra inversão de condicional pode ser usada na leitura de atributos dos objetos. É sempre indicado fazer uma afirmação em vez de uma negação. A seguir um exemplo:

Outras expressões negativas

```
class Documento {
  public boolean word() { // considerar uma regex :)
    return "doc".equals(extensaoArquivo)
        || "docx".equals(extensaoArquivo);
  }
// ...
// situação de uso:
public void imprime(Documento doc) {
  if (!doc.word()) {
    // apenas quando o documento não é do Word
// ...
```

Um leitor apressado poderia não ver o `!` e pensar que a lógica é executada quando o documento *é do Word*. É um detalhe, eu sei, mas tente evitar as negativas e experimente escrever um método mais claro, como no código refatorado a seguir:

Faça asserções afirmativas em vez de negativas

```
class Documento {
  public boolean isWord() {
    return "doc".equals(extension)
        || "docx".equals(extension);
  }
  // método novo
  public boolean isNotWord() { return ! isWord(); }
// ...
public void imprime(Documento doc) {
  if (doc.isNotWord()) {
    // apenas quando o documento não é do Word
// ...
```

Os condicionais afirmativos são, geralmente, mais intuitivos para quem está lendo o código. É uma refatoração simples e indolor, além disso demonstra capricho e cuidado.

Refatoração: introduzir constante

A refatoração Introduzir Constante é frequentemente usada para remover o *smell* números e strings mágicos. A mecânica de introdução de constante consiste em adicionar um membro privado (geralmente, mas pode ser público), estático e final, na classe onde se encontra o número (ou *string*) mágico e atribuir o valor, então o próximo passo é substituir todas as ocorrências deste valor mágico pela constante. A seguir um exemplo fácil:

Números mágicos, e agora Mr. M, o que significa doc.getStatus() != 2?

```
class Documento {
  // atributos
  public void imprime() {
    if (this.getStatus() != 2) return;
    if (this.getMargemDireita() > 400) {
      this.setMargemDireita(380);
// ...
```

Números e *strings* literais devem ser substituídos por constantes sempre que possível. A seguir o código anterior com os números mágicos substituídos por constantes:

Constantes fazem bem para a saúde

```
class Documento {
  private final static int PUBLICADO = 2;
  private final static int LIMITE_MARGEM_DIREITA = 400;
  private final static int MARGEM_DIREITA_MAXIMA = 380;
  // outros atributos
  public void imprime() {
    if (this.getStatus() != PUBLICADO) return;
    if (this.getMargemDireita() > LIMITE_MARGEM_DIREITA) {
        this.setMargemDireita(MARGEM_DIREITA_MAXIMA);
  }
// ...
```

Contantes também trazem outros benefícios, como isolar pontos de alteração, precisando mudar apenas a constante para atualizar todas as funcionalidades dependentes.

Refatoração: introduzir objeto parâmetro

Esta refatoração é utilizada em métodos com muitos parâmetros e/ou que permitam parâmetros nulos. Ela já foi abordada no tópico Projete para que `NULL` não seja passado como parâmetro, mas é importante abordá-la novamente. A mecânica consiste em criar uma classe e transformar os parâmetros do método em atributos dessa classe. A seguir um exemplo simples:

Parâmetros em excesso ou anuláveis dificultam a leitura

```
void imprime(Documento doc, TamanhoPagina tamanhoPagina,
             Integer nroCopias, Integer iniciaNaPagina,
             Orientacao orientacao) {
  // alguns parâmetros tem um valor padrão para quando forem nulos
  int copias = nroCopias == null ? 1 : nroCopias;
  int inicio = iniciaNaPagina == null ? 1 : iniciaNaPagina;
  Orientacao ori = orientacao == null ? Orientacao.RETRATO : orientacao;
// ...
// situação de uso:
printer.imprime(doc, TamanhoPagina.A4, null, null, null);
```

Chamadas de métodos que passam `null` como argumento não deixam claro o que está sendo omitido, precisando comparar com a assinatura do método para saber o que foi anulado. A seguir uma versão refatorada do código anterior:

Clarificando com um objeto parâmetro

```
void imprime(Documento doc, OpcoesImpressao opcoes) {
  // não é mais responsabilidade do método imprime
  // validar as opções de impressão,
  // as próprias opções "se validam"

  int copias = opcoes.getNumeroCopias();
  int inicio = opcoes.getPaginaInicial();
  Orientacao ori = opcoes.getOrientacaoPagina();
// ...

// opções de impressão extraídas
class OpcoesImpressao {
  TamanhoPagina tamanhoPagina;
  Integer nroCopias;
  Integer iniciaNaPagina;
  Orientacao orientacao

  int getNumeroCopias() {
    return nroCopias == null ? 1 : nroCopias;
  }
// ...

// situação de uso:
OpcoesImpressao opcoes = new OpcoesImpressao() {{
  orientacao = Orientacao.Retrato;
  tamanhoPagina = TamanhoPagina.CARTA;
}}; // hack para inicializar os atributos

printer.imprime(doc, opcoes);
```

Preste atenção à lógica que foi movida para a nova classe `OpcoesImpressao`, que define quais opções terão um *default*. Este tipo de refinamento no projeto também tem o propósito de esclarecer, mais precisamente, as responsabilidades de cada classe.

Finalizando, algumas linguagens suportam opcionalidade e a definição de um valor *default* para parâmetros, declarados na assinatura dos métodos (ou funções). A seguir um exemplo deste recurso no Ruby:

Valores default na linguagem Ruby: um código mais limpo

```
def imprime(doc, tamanho_pagina = Tamanho::A4, nro_copias = 1,
            pagina_inicial = 1, orientacao = Orientacao::RETRATO) {
# ...
# situação de uso:
printer.imprime(doc)
# os parâmetros omitidos receberiam seu valor default
```

Antes que alguém se magoe, outras linguagens contam com essa funcionalidade como: PHP, C#, Kotlin, Action Script, entre outras.

Programadores Profissionais programam em mais de uma linguagem.

Java está bem longe de ter sintaxe flexível e elegante, mas ainda parece mais palatável que C/C++. Na prática, todo programador profissional busca fazer o melhor com aquilo que tem. Programadores profissionais são pragmáticos, não são *fanboys* de certa linguagem, *framework*, editor, etc, embora certamente tenham uma preferência pessoal, que é deixada de lado quando querem ter o trabalho concluído, logo usam a ferramenta certa para o problema certo.

Se fosse...

Quando se esbarra num entrave complicado, uma das coisas que mais ouço, nas empresas e no ensino, começa com *"se fosse ..."*, *"ah, se fosse em Java teria um ..."*, *"ah, se fosse C seria mais ..."*, *"ah, se fosse até tal número"*, "*ah, se fosse desnecessário esse ponto e vírgula ...*". O programador deve fazer o melhor com o que ele tem e/ou construir ferramentas novas que minimizem os problemas. Só lamentar-se, sem tomar atitude, é uma atitude bem amadora.

Considerações sobre melhoria de código existente

Espero que após este capítulo tenhas entendido que um **bom** *software* não é aquele que funciona, mas aquele que funciona e é bem escrito (ou bem projetado). Todo este capítulo foi dedicado à qualidade interna do *software* – a parte do *software* que só os programadores têm acesso e condições de melhorar.

> ***"... agora até minha mãe consegue ler meus códigos ..."***
>
> Essas dicas de qualidade são abordadas no curso técnico. O momento mais incrível foi quando um estudante, no seu discurso de formatura, disse: *"agora até minha mãe consegue ler meus códigos"*. É um exagero, claro, mas **essa a intenção** - com as devidas proporções, é claro :)

Fechando, caso queiras te aprofundar mais em **refatoração** sugiro este *link* (EN) http://martinfowler.com/refactoring/, onde consta um catálogo de refatorações.

Este outro *link* (EN) http://users.csc.calpoly.edu/~jdalbey/305/Lectures/SmellsToRefactorings, tem uma referência muito boa sobre *smells* e suas refatorações sugeridas.

Capítulo `100` – Cenas dos próximos volumes

Esse livro foi criado a partir de uma compilação de trechos de um livro didático e de *slides* usados nas aulas da disciplina de Arquitetura e Projeto de Sistemas. Do que trata a disciplina, muitos temas ficaram de fora, tais como: Princípios e Padrões de Projeto, Princípios e Padrões de Arquitetura, Depuração e Teste de Software, Mapeamento Objeto/Relacional, etc. Esse livro seria enorme se eu abordasse o conteúdo inteiro, mas eu precisava torná-lo publicável, por isso ele abrange a **parte da implementação**, **por isso**: **Práticas**, **Técnicas e Refatorações de Código**.

Posso dizer que trabalharei para escrever novos volumes. O próximo será **O Programador Profissional**: **Princípios e Padrões de Projeto**.

Me alegraria muito saber que esse livro te foi útil, então estejas a vontade para me escrever. Espero te ter como leitor outra vez, seria uma honra!

Enfim, *adios*, um grande abraço e muito sucesso na sua carreira de **programador profissional**!
Cordialmente, Márcio Torres