Série especial da Addison-Wesley

"Qualquer tolo escreve um código que um computador possa entender. Bons programadores escrevem códigos que os seres humanos podem entender."
—M. Fowler (1999)

A Martin Fowler Signature Book

REFATORAÇÃO

Aperfeiçoando o design de códigos existentes

Martin Fowler
com contribuições de
Kent Beck

novatec

SEGUNDA EDIÇÃO

Refatoração

Aperfeiçoando o design de códigos existentes

Segunda Edição

Martin Fowler
com contribuições de Kent Beck

Addison-Wesley
Novatec

Editor: Rubens Prates
Tradução: Lúcia A. Kinoshita
Revisão gramatical: Tássia Carvalho
Editoração eletrônica: Carolina Kuwabata

ISBN do ebook: 978-85-7522-725-1

ISBN do impresso: 978-85-7522-724-4

Novatec Editora Ltda.
Rua Luís Antônio dos Santos 110
02460-000 – São Paulo, SP – Brasil
Tel.: +55 11 2959-6529
Email: novatec@novatec.com.br
Site: www.novatec.com.br
Twitter: twitter.com/novateceditora
Facebook: facebook.com/novatec
LinkedIn: linkedin.com/in/novatec

Para Cindy
– Martin

Sumário

Apresentação da primeira edição

A “refatoração” foi concebida nos círculos de Smalltalk, mas não demorou muito para que encontrasse seu caminho entre outras linguagens de programação. Como a refatoração é parte integrante do desenvolvimento de frameworks, o termo emerge rapidamente quando os desenvolvedores de framework falam de seu trabalho, surgindo quando eles detalham suas hierarquias de classes e quando se vangloriam da quantidade de linhas de código que conseguiram apagar. Os desenvolvedores de frameworks sabem que um framework não estará correto logo na primeira tentativa – ele deve evoluir à medida que ganharem experiência. Eles também sabem que o código será lido e modificado com mais frequência do que será escrito. O segredo para manter o código legível e modificável é a refatoração – para os frameworks, em particular, mas também para os softwares em geral.

Então, qual é o problema? Simplesmente este: a refatoração é arriscada. Ela exige mudanças que podem introduzir bugs sutis em um código que está funcionando. A refatoração, se não for feita de forma adequada, pode fazer você atrasar dias, até mesmo semanas. E a refatoração será mais arriscada ainda se for conduzida informalmente ou *ad hoc*. Você começa a explorar o código. Logo descobre novas oportunidades para alterá-lo, à medida que o explora com mais detalhes. Quanto mais você o analisa, mais detalhes percebe... e mais mudanças faz. Em algum momento, você cavará a própria sepultura e não conseguirá sair dela. Para evitar que isso aconteça, a refatoração deve ser feita sistematicamente. Quando meus coautores e eu escrevemos o livro *Padrões de projeto*, mencionamos que os padrões de projeto oferecem alvos para as refatorações. No entanto, identificar o alvo é somente uma parte do problema; transformar o código para atingir seus objetivos é outro desafio.

Martin Fowler e os autores que trabalham com ele fazem uma contribuição inestimável ao desenvolvimento de software orientado a objetos lançando uma luz sobre o processo de refatoração. Este livro explica os princípios e as melhores práticas para a refatoração, e mostra quando e onde você deve começar a explorar seu código a fim de aperfeiçoá-lo. Como parte essencial do livro, há um catálogo abrangente de refatorações. Cada refatoração

descreve a motivação e o mecanismo de uma transformação de código comprovada. Algumas das refatorações, por exemplo, *Extrair método* ou *Mover campo*, podem parecer óbvias.

Todavia, não se deixe enganar. Compreender o mecanismo de refatorações como essas é essencial para uma refatoração disciplinada. As refatorações deste livro ajudarão você a modificar o seu código, um pequeno passo de cada vez, reduzindo assim os riscos para a evolução de seu design. Você acrescentará rapidamente essas refatorações e seus nomes em seu vocabulário de desenvolvimento.

Minha primeira experiência com uma refatoração disciplinada, feita com "um passo de cada vez", ocorreu quando eu estava trabalhando com Kent Beck a aproximadamente 10 mil metros de altura. Ele garantiu que aplicássemos as refatorações do catálogo deste livro um passo de cada vez. Fiquei impressionado em ver como essa prática funcionava tão bem. Não só minha confiança no código resultante aumentou como também me senti menos estressado. Recomendo enfaticamente que você experimente usar essas refatorações: você e seu código se sentirão muito melhor.

— Erich Gamma, Object Technology International, Inc.
Janeiro de 1999

Prefácio

Era uma vez um consultor que visitou um projeto de desenvolvimento a fim de analisar um código que havia sido escrito. Enquanto observava a hierarquia de classes no centro do sistema, o consultor a achou bastante confusa. As classes de nível mais alto faziam certas pressuposições sobre como as classes funcionariam – suposições que estavam incorporadas no código herdado. Esse código, porém, não era apropriado a todas as subclasses, e era intensamente sobrescrito. Pequenas modificações na superclasse teriam reduzido bastante a necessidade de sobrescrevê-la. Em alguns lugares, uma intenção da superclasse não havia sido devidamente compreendida e o comportamento presente na superclasse estava duplicado. Em outros, diversas subclasses faziam o mesmo com um código que claramente poderia ser passado para um nível mais alto na hierarquia.

O consultor recomendou à gerência do projeto que o código fosse revisto e reorganizado – a gerência, porém, não ficou muito entusiasmada. O código parecia funcionar e havia pressões consideráveis no cronograma. Os gerentes disseram que eles fariam isso em algum momento no futuro.

O consultor também havia mostrado o que estava acontecendo aos programadores que trabalhavam com a hierarquia. Os programadores eram perspicazes e viram o problema. Sabiam que não era realmente um erro deles; às vezes, outro par de olhos se faz necessário para perceber o problema. Assim, os programadores passaram um ou dois dias reorganizando a hierarquia. Quando terminaram, eles haviam removido metade do código da hierarquia, sem reduzir a sua funcionalidade. Eles ficaram satisfeitos com o resultado e perceberam que havia ficado mais fácil e rápido adicionar novas classes e também as usar no restante do sistema.

A gerência do projeto não estava satisfeita. Os cronogramas eram apertados e havia muito trabalho a fazer. Esses dois programadores haviam gastado dois dias fazendo um trabalho que nada acrescentara às diversas funcionalidades que o sistema deveria disponibilizar em alguns meses. O código antigo funcionava bem. Sim, o design estava um pouco mais “puro” e um pouco mais “limpo”. No entanto, o projeto deveria disponibilizar um código que funcionasse, e não que satisfizesse a um acadêmico. O consultor

sugeriu que uma reorganização semelhante fosse feita em outras partes essenciais do sistema, o que faria o projeto ser interrompido por uma ou duas semanas. Tudo isso era para que o código tivesse um melhor aspecto, e não para que fizesse algo que ainda não estava fazendo.

O que você acha dessa história? Acha que o consultor estava certo em sugerir reorganizações adicionais? Ou você é adepto do velho ditado de engenharia que diz: “se está funcionando, não mexa”?

Devo admitir que sou um pouco tendencioso nesse caso. Eu era esse consultor. Seis meses depois, o projeto falhou, principalmente porque o código era complexo demais para depurar ou para ser ajustado de modo que tivesse um desempenho aceitável.

O consultor Kent Beck foi chamado para reiniciar o projeto – um exercício que envolveu a reescrita de quase todo o sistema, do zero. Ele fez muitas tarefas de modo diferente, mas uma das mudanças mais importantes foi insistir na reorganização contínua do código usando a refatoração. O aumento da eficácia da equipe e o papel desempenhado pela refatoração me inspiraram a escrever a primeira edição deste livro – desse modo, eu poderia transmitir o conhecimento que Kent e os demais haviam adquirido ao usar a refatoração para melhorar a qualidade do software.

Desde então, a refatoração vem sendo aceita como parte do vocabulário da programação. E o livro original teve boa aceitação. Apesar disso, dezoito anos é uma idade muito avançada para um livro de programação e, desse modo, achei que era hora de voltar e revisá-lo. Fazer isso me forçou a reescrever praticamente todas as páginas do livro. Porém, em certo sentido, pouca coisa mudou. A ideia principal da refatoração permanece a mesma; a maior parte das refatorações principais continuou essencialmente igual. Contudo, espero que a nova edição ajude mais pessoas a aprender como refatorar de modo eficaz.

O que é refatoração?

A refatoração é o processo de modificar um sistema de software de modo que não altere o comportamento externo do código, embora melhore a sua estrutura interna. É uma maneira disciplinada de reorganizar o código, minimizando as chances de introduzir bugs. Em sua essência, ao refatorar, você estará aperfeiçoando o design do código depois que ele foi escrito.

“Aperfeiçoar o design depois que ele foi escrito.” É uma frase inusitada. Durante boa parte da história do desenvolvimento de software, a maioria das

pessoas acreditava que deveríamos fazer o design antes, e, somente depois que ele estivesse pronto, escrever o código. Com o tempo, o código seria modificado, e a integridade do sistema – sua estrutura de acordo com esse design – entraria gradualmente em decadência. O código afundaria lentamente, passando da engenharia para o hacking.

A refatoração é o oposto dessa prática. Com ela, podemos partir de um design ruim, até mesmo caótico, e transformá-lo em um código bem estruturado. Cada passo é simples – até mesmo simplista. Passo um campo de uma classe para outra, extraio uma porção de código de um método para que ela tenha o próprio método ou desloco um código para cima ou para baixo em uma hierarquia. O efeito cumulativo dessas pequenas modificações pode melhorar radicalmente o design. É exatamente o inverso da noção de decadência de software.

Com a refatoração, o ponto de equilíbrio do trabalho muda. Percebo que o design, em vez de ser todo definido previamente, é feito de forma contínua durante o desenvolvimento. À medida que desenvolvo o sistema, aprendo a aperfeiçoar o design. O resultado dessa interação é um programa cujo design permanece bom enquanto o desenvolvimento continua.

O que este livro contém?

Este livro é um guia para a refatoração: foi escrito para um programador profissional. Meu objetivo é mostrar como refatorar de uma maneira controlada e eficiente. Você aprenderá a refatorar de modo a não introduzir bugs no código, e aperfeiçoará metodicamente a sua estrutura.

Tradicionalmente, um livro começa com uma introdução. Em princípio, concordo com isso, mas acho difícil apresentar a refatoração com uma discussão ou definições generalizadas – portanto, começarei com um exemplo. O Capítulo 1 parte de um pequeno programa com algumas falhas comuns de design e o refatora, transformando-o em um programa mais fácil de entender e de modificar. Você verá tanto o processo da refatoração quanto uma série de refatorações úteis. É o capítulo principal para ler se quiser entender realmente do que se trata a refatoração.

No Capítulo 2, abordarei mais dos princípios gerais da refatoração, darei algumas definições e os motivos para fazer uma refatoração. Além disso, apresentarei alguns dos desafios da refatoração. No Capítulo 3, Kent Beck me ajudou a descrever como identificar “maus cheiros” (bad smells) no código, e como eliminá-los com refatorações. Os testes desempenham um papel muito

importante na refatoração, de modo que o Capítulo 4 descreve como incluí-los no código.

O coração do livro – o catálogo de refatorações – ocupa o restante do volume. Embora esse catálogo, de forma alguma, seja um catálogo completo, ele inclui as refatorações essenciais de que a maioria dos desenvolvedores provavelmente precisará. Ele evoluiu a partir de anotações que fiz enquanto aprendia sobre refatoração no final dos anos 1990, e ainda as uso hoje em dia, pois não me lembro de todas. Quando quero fazer uma refatoração, por exemplo, *Separar em fases (Split Phase)*, o catálogo me lembra de como fazê-la de um modo seguro, passo a passo. Espero que essa seja a parte do livro à qual você recorrerá com frequência.

Exemplos com JavaScript

Como na maioria das áreas técnicas de desenvolvimento de software, exemplos de código são muito importantes para ilustrar os conceitos. No entanto, as refatorações são muito parecidas em diferentes linguagens. Ocasionalmente haverá detalhes específicos aos quais uma linguagem me forçará a prestar atenção, mas os elementos essenciais das refatorações serão os mesmos.

Escolhi JavaScript para ilustrar essas refatorações, pois achei que essa linguagem seria legível para o maior número de pessoas. Contudo, você não deverá ter dificuldades em adaptar as refatorações para qualquer linguagem que esteja usando no momento. Tentei não utilizar nenhuma das partes mais complicadas da linguagem, portanto você deverá entender as refatorações com um conhecimento apenas básico de JavaScript. Meu uso de JavaScript certamente não é um aval para a linguagem.

Embora eu use JavaScript em meus exemplos, isso não significa que as técnicas deste livro estejam limitadas a JavaScript. A primeira edição da obra usava Java, e muitos programadores o acharam conveniente, mesmo sem jamais terem escrito uma única classe Java. Cheguei a brincar com a ideia de ilustrar essa generalidade usando uma dúzia de linguagens diferentes nos exemplos, mas achei que seria confuso demais para o leitor. Este livro foi escrito para programadores de qualquer linguagem. Exceto pelas seções de exemplo, não faço nenhuma pressuposição sobre a linguagem. Espero que o leitor absorva meus comentários gerais e os aplique à linguagem que estiver usando. Com efeito, espero que os leitores tomem os exemplos em JavaScript e os adaptem à sua linguagem.

Isso significa que, exceto nas ocasiões em que discuto exemplos específicos, quando falo de “classe”, “módulo”, “função” etc., uso esses termos no sentido geral da programação, e não como termos específicos do modelo da linguagem JavaScript.

O fato de estar usando JavaScript como linguagem para os exemplos também significa que procuro evitar estilos dessa linguagem que sejam menos conhecidos daqueles que não sejam programadores habituais de JavaScript. Este não é um livro sobre “refatoração em JavaScript” – é um livro sobre refatoração em geral, que, por acaso, usa JavaScript. Há muitas refatorações interessantes específicas para JavaScript (por exemplo, refatorações de callbacks, para promises, para async/await), mas elas estão fora do escopo deste livro.

Quem deve ler este livro?

O livro está voltado para um programador profissional – alguém que escreve software para viver. Os exemplos e as discussões incluem bastante código para ler e entender, e estão em JavaScript, mas devem ser aplicáveis à maioria das linguagens. A expectativa é que um programador tenha certa experiência para apreciar o que está acontecendo no livro, mas não pressuponho que tenha muito conhecimento.

Embora o alvo principal deste livro seja um desenvolvedor que queira aprender sobre refatoração, a obra também é importante para alguém que já a conheça – ele pode ser usado como um recurso para o ensino. Neste livro, investi muito esforço em explicar como diversas refatorações funcionam, portanto um desenvolvedor experiente poderá usar este material para orientador os colegas.

Apesar de ter o código como foco, a refatoração exerce um impacto significativo no design do sistema. É essencial que designers e arquitetos mais experientes entendam os princípios da refatoração e os usem em seus projetos. A refatoração será apresentada de forma mais conveniente se for feita por um desenvolvedor respeitado e experiente. Um desenvolvedor como esse é capaz de compreender melhor os princípios por trás da refatoração e adaptá-los em um local de trabalho específico. Isso será particularmente verdadeiro se você usar uma linguagem diferente de JavaScript, pois será necessário adaptar os exemplos apresentados para outras linguagens.

Eis o modo de tirar o máximo proveito deste livro sem lê-lo por completo.

- Se quiser entender o que é refatoração, leia o Capítulo 1 – o exemplo deverá deixar o processo claro.
- Se quiser entender por que deve refatorar, leia os dois primeiros capítulos. Eles mostram o que é a refatoração e por que você deve usá-la.
- Se quiser descobrir onde deve refatorar, leia o Capítulo 3. Ele mostra os sinais que sugerem a necessidade de refatoração.
- Se quiser realmente refatorar, leia os quatro primeiros capítulos completos e, em seguida, vá direto para o catálogo. Leia o suficiente do catálogo para saber, grosso modo, o que há nele. Você não precisa entender todos os detalhes. Quando tiver realmente que fazer uma refatoração, leia os detalhes sobre ela e utilize-a como ajuda. O catálogo é uma seção de referência, portanto é provável que você não queira lê-lo de uma só vez.

Uma parte importante na escrita deste livro foi nomear as diversas refatorações. A terminologia nos ajuda na comunicação, de modo que, quando um desenvolvedor aconselha outro a extrair um código para uma função ou separar algum processamento em fases distintas, ambos compreenderão as referências a *Extrair função (Extract Function)* e a *Separar em fases (Split Phase)*. Esse vocabulário também ajuda na seleção de refatorações automatizadas.

Construindo sobre uma base lançada por outros

Devo dizer logo de imediato que tenho uma grande dívida com este livro – uma dívida com aqueles cujo trabalho nos anos 1990 foi responsável pelo desenvolvimento da refatoração. Foi aprendendo com a experiência dessas pessoas que me inspirei e adquiri conhecimentos para escrever a primeira edição deste livro, e, embora muitos anos tenham se passado, é importante que eu continue reconhecendo as bases que eles lançaram. O ideal é que um deles tivesse escrito a primeira edição, mas acabei sendo a pessoa com o tempo e a energia para fazê-lo.

Dois dos principais proponentes iniciais da refatoração foram Ward Cunningham e Kent Beck. Eles foram os primeiros a usá-la como base para o desenvolvimento e adaptaram seus processos para tirar proveito dela. Em particular, foi meu trabalho em parceria com Kent que me mostrou a importância da refatoração – uma inspiração que resultou diretamente neste

livro.

Ralph Johnson lidera um grupo na Universidade de Illinois em Urbana-Champaign que se destaca pelas contribuições práticas à tecnologia de orientação a objetos. Ralph tem sido um líder no campo da refatoração há muito tempo, e vários de seus alunos foram responsáveis pelos primeiros trabalhos essenciais nessa área. Bill Opdyke desenvolveu o primeiro trabalho escrito e detalhado sobre refatoração em sua tese de doutorado. John Brant e Don Roberts foram além das palavras escritas – eles criaram a primeira ferramenta de refatoração automatizada, o Refactoring Browser, para refatorar programas Smalltalk.

Muitas pessoas fizeram a área de refatoração progredir desde a primeira edição deste livro. Em particular, o trabalho daqueles que adicionaram refatorações automatizadas nas ferramentas de desenvolvimento contribuiu bastante para facilitar a vida dos programadores. É fácil para mim aceitar que posso renomear gratuitamente uma função bastante usada com uma simples sequência de teclas – mas essa facilidade se deve aos esforços das equipes de IDEs cujo trabalho ajuda a todos nós.

Agradecimentos

Mesmo com toda aquela pesquisa na qual pude me basear, ainda precisei de muita ajuda para escrever este livro. A primeira edição contou intensamente com a experiência e o incentivo de Kent Beck. Ele foi o primeiro a apresentar a refatoração para mim, inspirando-me a fazer anotações para registrar as refatorações e ajudando-me a transformá-las em prosa definitiva. Kent concebeu a ideia de Maus Cheiros (Code Smells) no código. Frequentemente acho que ele teria escrito uma primeira edição melhor que a minha – se não estivesse escrevendo o livro básico sobre Extreme Programming.

Todos os autores de livros técnicos que conheço mencionam a grande dívida que têm com os revisores técnicos. Todos nós já escrevemos trabalhos com grandes imperfeições que só foram identificadas por nossos colegas atuando como revisores. Eu mesmo não faço muita revisão técnica, em parte porque não me acho muito bom nisso, de modo que tenho profunda admiração por aqueles que assumem essa tarefa. Não há a mínima recompensa por revisar o livro de outra pessoa, portanto fazer isso é um grande ato de generosidade.

Quando comecei a trabalhar seriamente no livro, criei uma lista de discussão com conselheiros que pudessem me dar feedback. À medida que

fazia progressos, eu enviava rascunhos de novos conteúdos para esse grupo e solicitava seu feedback. Gostaria de agradecer às pessoas a seguir por postarem seus feedbacks na lista de discussão: Arlo Belshee, Avdi Grimm, Beth Anders-Beck, Bill Wake, Brian Guthrie, Brian Marick, Chad Wathington, Dave Farley, David Rice, Don Roberts, Fred George, Giles Alexander, Greg Doench, Hugo Corbucci, Ivan Moore, James Shore, Jay Fields, Jessica Kerr, Joshua Kerievsky, Kevlin Henney, Luciano Ramalho, Marcos Brizeno, Michael Feathers, Patrick Kua, Pete Hodgson, Rebecca Parsons e Trisha Gee.

Nesse grupo, gostaria de enfatizar particularmente a ajuda em especial que obtive de Beth Anders-Beck, James Shore e Pete Hodgson com JavaScript.

Depois que tinha uma versão preliminar bem completa, enviei-a para outros revisores porque queria que outros olhos lessem o rascunho como um todo. William Chargin e Michael Hunger ofereceram comentários extremamente detalhados na revisão. Também recebi muitos comentários importantes de Bob Martin e Scott Davis. Bill Wake acrescentou suas contribuições à lista de discussão fazendo uma revisão completa da primeira versão preliminar.

Meus colegas na ThoughtWorks são uma fonte constante de ideias e de feedback para minha escrita. Houve inúmeras perguntas, comentários e observações que contribuíram para que eu pensasse no livro e o escrevesse. Um dos melhores aspectos de ser um funcionário da ThoughtWorks é que eles me permitem investir um tempo considerável na escrita. Em particular, agradeço as conversas constantes e as ideias que recebi de Rebecca Parsons, nossa CTO.

Na Pearson, Greg Doench é meu editor de aquisição, enfrentando muitos problemas para conduzir um livro até a sua publicação. Julie Nahil é minha editora de produção. Fiquei feliz em trabalhar novamente com Dmitry Kirsanov para revisão gramatical e com Alina Kirsanova para composição e indexação.

CAPÍTULO 1

Refatoração: primeiro exemplo

Como começar a falar de refatoração? O modo tradicional seria apresentar o histórico sobre o assunto, os princípios básicos e informações desse tipo. Quando alguém faz isso em uma conferência, fico um pouco sonolento. Minha mente começa a divagar, e mantenho um processo de baixa prioridade em segundo plano fazendo polling no palestrante até que um exemplo seja apresentado.

Os exemplos me acordam, pois posso ver o que está acontecendo. Com os princípios, é muito fácil fazer generalizações amplas – e é muito difícil saber como aplicá-los. Um exemplo ajuda a deixar tudo claro.

Assim, darei início a este livro com um exemplo de refatoração. Explicarei como a refatoração funciona e darei a você uma noção sobre o processo de refatoração. Poderei então fazer a introdução costumeira, no estilo dos princípios, no próximo capítulo.

Com um exemplo introdutório, porém, vejo-me diante de um problema. Se escolho um programa longo e o descrevo, mostrando como ele é fatorado, será complicado demais para um leitor mortal me acompanhar. (Tentei fazer isso no livro original – e acabei jogando fora dois exemplos que, apesar de serem bem pequenos, ocupavam umas cem páginas cada um para serem descritos.) No entanto, se escolher um programa suficientemente pequeno para que seja compreensível, teremos a impressão de que a refatoração não vale a pena.

Desse modo, encontro-me no clássico dilema de qualquer pessoa que queira descrever técnicas úteis para os programas do mundo real. Falando francamente, o esforço para fazer toda a refatoração que mostrarei a você no pequeno programa que usaremos não compensa. Contudo, se o código que será apresentado fizer parte de um sistema maior, a refatoração será importante. Olhe para meu exemplo e imagine-o no contexto de um sistema muito maior.

Ponto de partida

Na primeira edição deste livro, meu programa inicial exibia uma conta de uma videolocadora, e isso, atualmente, poderia levar muitos de vocês a perguntar: “O que é uma videolocadora?”. Em vez de responder a essa pergunta, substituí o exemplo por algo mais antigo, porém, ao mesmo tempo, atual.

Pense em uma companhia de atores de teatro que saia para participar de vários eventos apresentando suas peças. Em geral, os clientes solicitarão algumas peças e a companhia cobrará deles com base no número de espectadores e no tipo de peça encenada. Atualmente há dois tipos de peças que a companhia apresenta: tragédias e comédias. Além de apresentar uma conta pela apresentação, a companhia dá “créditos por volume” aos seus clientes, os quais podem ser usados como descontos em futuras apresentações – pense nisso como um mecanismo de fidelização do cliente.

Os atores armazenam dados sobre suas peças em um arquivo JSON simples semelhante a este:

plays.json…

```
{
  "hamlet": {"name": "Hamlet", "type": "tragedy"},
  "as-like": {"name": "As You Like It", "type": "comedy"},
  "othello": {"name": "Othello", "type": "tragedy"}
}
```

Os dados para as contas também estão em um arquivo JSON:

invoices.json…

```
[
  {
    "customer": "BigCo",
    "performances": [
      {
        "playID": "hamlet",
        "audience": 55
      },
      {
        "playID": "as-like",
        "audience": 35
      },
      {
        "playID": "othello",
```

```
      "audience": 40
    }
  ]
}
]
```

O código que exibe a conta está na função simples a seguir:

```
function statement (invoice, plays) {
  let totalAmount = 0;
  let volumeCredits = 0;
  let result = `Statement for ${invoice.customer}\n`;
  const format = new Intl.NumberFormat("en-US",
                        { style: "currency", currency: "USD",
                          minimumFractionDigits: 2 }).format;

  for (let perf of invoice.performances) {
    const play = plays[perf.playID];
    let thisAmount = 0;

    switch (play.type) {
    case "tragedy":
      thisAmount = 40000;
      if (perf.audience > 30) {
        thisAmount += 1000 * (perf.audience - 30);
      }
      break;
    case "comedy":
      thisAmount = 30000;
      if (perf.audience > 20) {
        thisAmount += 10000 + 500 * (perf.audience - 20);
      }
      thisAmount += 300 * perf.audience;
      break;
    default:
        throw new Error(`unknown type: ${play.type}`);
    }

    // soma créditos por volume
    volumeCredits += Math.max(perf.audience - 30, 0);
    // soma um crédito extra para cada dez espectadores de comédia
    if ("comedy" === play.type) volumeCredits += Math.floor(perf.audience / 5);

    // exibe a linha para esta requisição
    result += ` ${play.name}: ${format(thisAmount/100)} (${perf.audience} seats)\n`;
    totalAmount += thisAmount;
  }
```

```
  result += `Amount owed is ${format(totalAmount/100)}\n`;
  result += `You earned ${volumeCredits} credits\n`;
  return result;
}
```

A execução desse código nos arquivos de dados de teste anteriores resulta na seguinte saída:

```
Statement for BigCo
  Hamlet: $650.00 (55 seats)
  As You Like It: $580.00 (35 seats)
  Othello: $500.00 (40 seats)
Amount owed is $1,730.00
You earned 47 credits
```

Comentários sobre o programa inicial

Quais são suas ideias acerca do design desse programa? A primeira afirmação que eu faria é que ele é tolerável como está – um programa tão pequeno que não exige nenhuma estrutura profunda para ser compreensível. Lembre-se, porém, do que eu disse antes sobre a necessidade de manter os exemplos concisos. Pense nesse programa em uma escala maior – talvez com centenas de linhas de extensão. Com esse tamanho, uma única função inline seria difícil de entender.

Considerando que o programa funciona, qualquer afirmação sobre a sua estrutura não seria apenas um julgamento estético, ou uma demonstração de desgosto por um código “feio”? Afinal de contas, o compilador não se importa se o código é feio ou claro. Todavia, se altero o sistema, há um ser humano envolvido, e os seres humanos se importam. Um sistema com design ruim é difícil de ser alterado – porque é difícil identificar o que deve ser modificado e como essas modificações interagirão com o código existente para termos o comportamento desejado. E se for difícil identificar o que deve ser alterado, há uma boa chance de que vou cometer erros e introduzir bugs.

Desse modo, se eu tiver de modificar um programa com centenas de linhas de código, preferiria que ele estivesse estruturado na forma de um conjunto de funções e de outros elementos de programa que me permitissem compreender mais facilmente o que o programa faz. Se o programa não estiver estruturado, em geral será mais fácil para mim conferir-lhe uma estrutura antes, e então fazer a alteração necessária.

Se você tiver de acrescentar uma funcionalidade em um programa, mas o

código não está estruturado de modo conveniente, refatore-o antes para que seja mais fácil acrescentar a funcionalidade, e então a acrescente.

Nesse exemplo, tenho duas modificações que os usuários gostariam de fazer. Em primeiro lugar, eles querem que o demonstrativo seja exibido em HTML. Considere o impacto que essa modificação teria. Estou diante de uma situação que exigiria colocar instruções condicionais extras em torno de cada instrução que acrescente uma string no resultado. Isso adicionará uma série de complexidades à função. Diante disso, a maioria das pessoas preferirá copiar o método e alterá-lo para que gere HTML. Fazer uma cópia não parece uma tarefa muito custosa, mas criará todo tipo de problemas no futuro. Qualquer mudança na lógica de cobrança me forçaria a atualizar os dois métodos – e garantir que sejam atualizados de forma consistente. Se eu estivesse escrevendo um programa que não mudasse nunca, esse tipo de operação de copiar e colar não seria um problema. Porém, se for um programa com vida útil longa, a duplicação será então uma ameaça.

Isso me remete à segunda modificação. Os atores querem encenar outros tipos de peças: eles esperam acrescentar os estilos histórico, pastoral, pastoral-cômico, histórico-pastoral, trágico-histórico, trágico-cômico-histórico-pastoral, cenas indivisíveis e poema ilimitado ao seu repertório. Eles ainda não decidiram exatamente o que querem fazer nem quando. Essa mudança afetará tanto o modo como suas peças serão cobradas quanto a forma de calcular os créditos por volume. Como desenvolvedor experiente, posso garantir que, qualquer que seja o esquema concebido, eles o modificarão novamente no período de seis meses. Afinal de contas, quando chegam requisições por funcionalidades, elas não chegam como espiões solitários, mas em batalhões.

Novamente, é naquele método `statement` em que as alterações devem ser feitas para lidar com mudanças nas regras de classificação e de cobrança. No entanto, se eu copiasse `statement` para `htmlStatement`, seria necessário garantir que qualquer modificação fosse consistente. Além do mais, à medida que as regras se tornarem mais complexas, será mais difícil identificar os locais em que as modificações devem ser feitas, e mais difícil efetuá-las sem cometer um erro.

Deixe-me enfatizar que são essas mudanças que determinam a necessidade de fazer uma refatoração. Se o código estiver funcionando e não tiver de ser alterado, deixá-lo como está não é um problema. Seria bom aperfeiçoá-lo, mas, a menos que alguém precise entendê-lo, ele não estará causando

nenhum dano real. Contudo, assim que alguém precisar entender como esse código funciona e tiver dificuldade para saber o que ele faz, será necessário tomar alguma atitude a respeito.

Primeiro passo na refatoração

Toda vez que faço uma refatoração, o primeiro passo é sempre o mesmo. Devo garantir que tenho um conjunto robusto de testes para essa seção de código. Os testes são essenciais porque, apesar de fazer uma refatoração estruturada a fim de evitar a maior parte das oportunidades para introdução de bugs, ainda sou um ser humano e cometo erros. Quanto maior o programa, mais provável será que minhas alterações, inadvertidamente, façam algo deixar de funcionar – na era digital, o nome para a fragilidade é software.

Como statement devolve uma string, o que faço é criar algumas faturas, associo a cada uma delas algumas apresentações de vários tipos de peças de teatro e gero as strings do demonstrativo. Faço então uma comparação de strings entre a nova string e algumas strings de referência verificadas manualmente. Crio todos esses testes usando um framework de testes para que eu possa executá-los somente pressionando uma tecla em meu ambiente de desenvolvimento. Os testes demoram apenas alguns segundos para executar e, como você verá, eu os executo com frequência.

Uma parte importante dos testes é o modo como eles apresentam os resultados. São exibidos com verde, o que significa que todas as strings são idênticas às strings de referência, ou com vermelho, mostrando uma lista de falhas – as linhas cujos resultados foram diferentes. Os testes, portanto, são conferidos automaticamente. É essencial criar testes que sejam conferidos automaticamente. Se eu não fizesse isso, acabaria gastando tempo para verificar manualmente os valores dos testes, comparando-os com valores anotados em um bloquinho, e isso me causaria atrasos. Frameworks de teste modernos oferecem todas as funcionalidades necessárias para escrever e executar testes conferidos automaticamente.

Antes de começar a refatorar, certifique-se de que você tenha um conjunto de testes robusto. Esses testes devem ser conferidos automaticamente.

À medida que fizer a refatoração, contarei com os testes. Penso neles como um detector de bugs para me proteger contra meus próprios erros. Ao escrever o que quero duas vezes, no código e no teste, eu teria de cometer o

erro de forma consistente nos dois lugares para enganar o detector. Ao conferir meu trabalho duas vezes, reduzo as chances de fazer algo errado. Embora criar testes exija tempo, acabo economizando esse tempo, com juros consideráveis, ao gastar menos tempo na depuração. Essa é uma parte tão importante da refatoração que dedicarei um capítulo inteiro a ela (Capítulo 4, Escrevendo testes).

Decompondo a função statement

Ao refatorar uma função longa como essa, tento identificar mentalmente os pontos que separam diferentes partes do comportamento geral. A primeira porção que me salta aos olhos é a instrução switch no meio.

```
function statement (invoice, plays) {
  let totalAmount = 0;
  let volumeCredits = 0;
  let result = `Statement for ${invoice.customer}\n`;
  const format = new Intl.NumberFormat("en-US",
                       { style: "currency", currency: "USD",
                         minimumFractionDigits: 2 }).format;

  for (let perf of invoice.performances) {
    const play = plays[perf.playID];
    let thisAmount = 0;

    switch (play.type) {
    case "tragedy":
      thisAmount = 40000;
      if (perf.audience > 30) {
        thisAmount += 1000 * (perf.audience - 30);
      }
      break;
    case "comedy":
      thisAmount = 30000;
      if (perf.audience > 20) {
        thisAmount += 10000 + 500 * (perf.audience - 20);
      }
      thisAmount += 300 * perf.audience;
      break;
    default:
        throw new Error(`unknown type: ${play.type}`);
    }

    // soma créditos por volume
```

```
    volumeCredits += Math.max(perf.audience - 30, 0);
    // soma um crédito extra para cada dez espectadores de comédia
    if ("comedy" === play.type) volumeCredits += Math.floor(perf.audience / 5);

    // exibe a linha para esta requisição
    result += ` ${play.name}: ${format(thisAmount/100)} (${perf.audience} seats)\n`;
    totalAmount += thisAmount;
  }
  result += `Amount owed is ${format(totalAmount/100)}\n`;
  result += `You earned ${volumeCredits} credits\n`;
  return result;
}
```

Enquanto observo essa parte, concluo que ela calcula o valor cobrado para uma apresentação. Essa conclusão é um insight sobre o código. Porém, como afirma Ward Cunningham, essa compreensão está em minha mente – uma forma de armazenagem reconhecidamente volátil. Tenho de persisti-la, passando-a de minha mente de volta para o código. Desse modo, caso eu retorne ao código mais tarde, ele me dirá o que está fazendo – não terei de descobrir novamente.

O modo de colocar essa compreensão no código é transformar essa porção de código em uma função própria, nomeando-a com base no que ela faz – algo como `amountFor(aPerformance)`. Quando quero transformar uma porção de código em uma função dessa maneira, tenho um procedimento para isso que minimiza as chances de fazer algo errado. Escrevi esse procedimento, e, para que fosse mais fácil referenciá-lo, chamei-o de *Extrair função (Extract Function)*.

Em primeiro lugar, preciso observar o fragmento em busca de qualquer variável que não estará mais no escopo depois que eu tiver extraído o código em sua própria função. Nesse caso, tenho três variáveis: `perf`, `play` e `thisAmount`. As duas primeiras são usadas pelo código extraído, mas não são modificadas, portanto posso passá-las como parâmetros. Variáveis modificadas exigem mais atenção. Nesse caso, há apenas uma, portanto posso devolvê-la. Também posso levar sua inicialização para dentro do código extraído. Tudo isso resulta no código a seguir:

function statement…

```
function amountFor(perf, play) {
  let thisAmount = 0;
  switch (play.type) {
  case "tragedy":
```

```
      thisAmount = 40000;
      if (perf.audience > 30) {
        thisAmount += 1000 * (perf.audience - 30);
      }
      break;
    case "comedy":
      thisAmount = 30000;
      if (perf.audience > 20) {
        thisAmount += 10000 + 500 * (perf.audience - 20);
      }
      thisAmount += 300 * perf.audience;
      break;
    default:
        throw new Error(`unknown type: ${play.type}`);
    }
    return thisAmount;
  }
```

Quando uso um cabeçalho como "*function algumNome…*" em itálico em algum código, significa que o código que se segue está dentro do escopo da função, do arquivo ou da classe cujo nome está no cabeçalho. Em geral, haverá mais código nesse , mas ele não será mostrado, pois não estará sendo discutido na ocasião.

O código original de statement agora chama essa função para atribuir valor a thisAmount:

nível mais alto…

```
  function statement (invoice, plays) {
    let totalAmount = 0;
    let volumeCredits = 0;
    let result = `Statement for ${invoice.customer}\n`;
    const format = new Intl.NumberFormat("en-US",
                          { style: "currency", currency: "USD",
                            minimumFractionDigits: 2 }).format;
    for (let perf of invoice.performances) {
      const play = plays[perf.playID];
      let thisAmount = amountFor(perf, play);

      // soma créditos por volume
      volumeCredits += Math.max(perf.audience - 30, 0);
      // soma um crédito extra para cada
      // dez espectadores de comédia
      if ("comedy" === play.type) volumeCredits += Math.floor(perf.audience / 5);
```

```
    // exibe a linha para esta requisição
    result += ` ${play.name}: ${format(thisAmount/100)} (${perf.audience} seats)\n`;
    totalAmount += thisAmount;
  }
  result += `Amount owed is ${format(totalAmount/100)}\n`;
  result += `You earned ${volumeCredits} credits\n`;
  return result;
```

Depois de fazer essa alteração, compilo e testo imediatamente para ver se causei alguma falha. Testar após cada refatoração é um hábito importante, embora simples. Erros são fáceis de cometer – pelo menos, acho que são. Testar depois de cada alteração significa que, quando cometo um erro, terei apenas uma pequena modificação a ser considerada a fim de identificar o erro, fazendo com que seja muito mais fácil localizá-lo e corrigi-lo. Essa é a essência do processo de refatoração: pequenas modificações e testes depois de cada modificação. Se eu tentar fazer muitas delas, cometer um erro me forçará a um episódio complexo de depuração que poderá consumir bastante tempo. Pequenas modificações, que permitam um ciclo de feedback rápido, são o segredo para evitar essa confusão.

Quando uso *compilar*, quero dizer fazer o que for necessário para deixar o JavaScript executável. Como JavaScript é diretamente executável, isso pode significar não fazer nada; em outros casos, porém, pode ser mover o código para um diretório de saída e/ou usar um processador como o Babel [babel].

A refatoração altera os programas em passos pequenos, de modo que, se você cometer um erro, será fácil localizar o bug.

Por ser um código JavaScript, posso extrair `amountFor` e colocá-lo em uma função aninhada de `statement`. É conveniente, pois significa que não preciso passar dados que estão no escopo da função que a contém para a função que acabou de ser extraída. Nesse exemplo, isso não fará diferença, mas é um problema a menos para tratar.

Em nosso caso, os testes passaram, portanto, meu próximo passo é fazer commit da alteração em meu sistema local de controle de versões. Uso um sistema de controle de versões, por exemplo, git ou mercurial, que me permita fazer commits privados. Faço commit depois de cada refatoração bem-sucedida, de modo que eu possa retornar facilmente a um estado funcional caso me atrapalhe no futuro. Então, reúno as alterações em commits mais significativos antes de enviá-las para um repositório

compartilhado.

Extrair função (Extract Function) é uma refatoração comum para ser automatizada. Se eu estivesse programando em Java, teria instintivamente usado a sequência de teclas em meu IDE para fazer essa refatoração. Quando escrevi este livro, não havia um suporte tão robusto para essa refatoração nas ferramentas JavaScript, portanto tive de fazer isso manualmente. Não é difícil, embora eu precise ter cuidado com as variáveis de escopo local.

Depois de ter usado *Extrair função (Extract Function)*, observo o código que extraí para ver se há algo rápido e fácil que eu possa fazer a fim de dar mais clareza à função extraída. Minha primeira tarefa é renomear algumas das variáveis para deixá-las mais claras, por exemplo, alterar `thisAmount` para `result`.

function statement…

```
function amountFor(perf, play) {
  let result = 0;
  switch (play.type) {
  case "tragedy":
    result = 40000;
    if (perf.audience > 30) {
      result += 1000 * (perf.audience - 30);
    }
    break;
  case "comedy":
    result = 30000;
    if (perf.audience > 20) {
      result += 10000 + 500 * (perf.audience - 20);
    }
    result += 300 * perf.audience;
    break;
  default:
      throw new Error(`unknown type: ${play.type}`);
  }
  return result;
}
```

Chamar sempre o valor de retorno de uma função de “result” faz parte do meu padrão de programação. Desse modo, sempre saberei qual é o seu papel. Novamente, compilo, testo e faço commit. Em seguida, passo para o primeiro argumento.

function statement…

```
function amountFor(aPerformance, play) {
  let result = 0;
  switch (play.type) {
  case "tragedy":
    result = 40000;
    if (aPerformance.audience > 30) {
      result += 1000 * (aPerformance.audience - 30);
    }
    break;
  case "comedy":
    result = 30000;
    if (aPerformance.audience > 20) {
      result += 10000 + 500 * (aPerformance.audience - 20);
    }
    result += 300 * aPerformance.audience;
    break;
  default:
      throw new Error(`unknown type: ${play.type}`);
  }
  return result;
}
```

Mais uma vez, estamos seguindo meu estilo de programação. Com uma linguagem dinamicamente tipada como JavaScript, é conveniente manter o controle dos tipos – desse modo, meu nome default para um parâmetro inclui o nome do tipo. Uso um artigo indefinido para ele, a menos que haja alguma informação específica sobre o seu papel que deva ser incluída no nome. Aprendi essa convenção com Kent Beck [Beck SBPP] e continuo achando-a útil.

> *Qualquer tolo consegue escrever códigos que um computador possa entender. Bons programadores escrevem códigos que os seres humanos podem entender.*

O esforço de renomear variáveis vale a pena? Com certeza. Um bom código deve comunicar claramente o que faz, e nomes de variáveis são essenciais para a clareza de um código. Nunca tenha medo de mudar nomes para ter mais clareza. Com boas ferramentas para localizar e substituir, em geral isso não será difícil; testes e tipagem estática em uma linguagem que a aceita darão destaque a qualquer ocorrência que você tenha deixado passar. Além disso, com ferramentas automatizadas de refatoração, é trivial renomear até mesmo funções amplamente usadas.

O próximo item a ser considerado para renomear é o parâmetro play, mas reservo um destino diferente para ele.

Removendo a variável play

Enquanto considero os parâmetros de amountFor, observo de onde eles vêm. aPerformance é proveniente da variável do laço, portanto mudará naturalmente a cada iteração. Entretanto, play é obtido da apresentação, portanto não é necessário passá-lo como parâmetro – posso simplesmente o recalcular em amountFor. Quando divido uma função longa, gosto de me livrar de variáveis como play, pois variáveis temporárias criam muitos nomes com escopo local, complicando as extrações. A refatoração que usarei nesse caso é *Substituir variável temporária por consulta (Replace Temp with Query)*.

Começo extraindo o lado direito da atribuição, colocando-o em uma função.

function statement…

```
function playFor(aPerformance) {
  return plays[aPerformance.playID];
}
```

nível mais alto…

```
function statement (invoice, plays) {
  let totalAmount = 0;
  let volumeCredits = 0;
  let result = `Statement for ${invoice.customer}\n`;
  const format = new Intl.NumberFormat("en-US",
                       { style: "currency", currency: "USD",
                         minimumFractionDigits: 2 }).format;
  for (let perf of invoice.performances) {
    const play = playFor(perf);
    let thisAmount = amountFor(perf, play);

    // soma créditos por volume
    volumeCredits += Math.max(perf.audience - 30, 0);
    // soma um crédito extra para cada
    // dez espectadores de comédia
    if ("comedy" === play.type) volumeCredits += Math.floor(perf.audience / 5);

    // exibe a linha para esta requisição
    result += ` ${play.name}: ${format(thisAmount/100)} (${perf.audience} seats)\n`;
    totalAmount += thisAmount;
  }
  result += `Amount owed is ${format(totalAmount/100)}\n`;
```

```
  result += `You earned ${volumeCredits} credits\n`;
  return result;
```

Compilo-testo-faço commit, e então uso *Internalizar variável (Inline Variable)*.

nível mais alto…

```
function statement (invoice, plays) {
  let totalAmount = 0;
  let volumeCredits = 0;
  let result = `Statement for ${invoice.customer}\n`;
  const format = new Intl.NumberFormat("en-US",
                        { style: "currency", currency: "USD",
                          minimumFractionDigits: 2 }).format;

  for (let perf of invoice.performances) {
    const play = playFor(perf);
    let thisAmount = amountFor(perf, playFor(perf));

    // soma créditos por volume
    volumeCredits += Math.max(perf.audience - 30, 0);
    // soma um crédito extra para cada
    // dez espectadores de comédia
    if ("comedy" === playFor(perf).type) volumeCredits += Math.floor(perf.audience / 5);

    // exibe a linha para esta requisição
    result += ` ${playFor(perf).name}: ${format(thisAmount/100)} (${perf.audience} seats)\n`;
    totalAmount += thisAmount;
  }
  result += `Amount owed is ${format(totalAmount/100)}\n`;
  result += `You earned ${volumeCredits} credits\n`;
  return result;
```

Compilo-testo-faço commit. Com esse código internalizado, posso então aplicar *Mudar declaração de função (Change Function Declaration)* em `amountFor` para remover o parâmetro `play`. Faço isso em dois passos. Em primeiro lugar, uso a nova função em `amountFor`.

function statement…

```
function amountFor(aPerformance, play) {
  let result = 0;
  switch (playFor(aPerformance).type) {
  case "tragedy":
    result = 40000;
    if (aPerformance.audience > 30) {
```

```
        result += 1000 * (aPerformance.audience - 30);
      }
      break;
    case "comedy":
      result = 30000;
      if (aPerformance.audience > 20) {
        result += 10000 + 500 * (aPerformance.audience - 20);
      }
      result += 300 * aPerformance.audience;
      break;
    default:
        throw new Error(`unknown type: ${playFor(aPerformance).type}`);
    }
    return result;
  }
```

Compilo-testo-faço commit, e então removo o parâmetro.

nível mais alto…

```
  function statement (invoice, plays) {
    let totalAmount = 0;
    let volumeCredits = 0;
    let result = `Statement for ${invoice.customer}\n`;
    const format = new Intl.NumberFormat("en-US",
                          { style: "currency", currency: "USD",
                            minimumFractionDigits: 2 }).format;
    for (let perf of invoice.performances) {
      let thisAmount = amountFor(perf~~, playFor(perf)~~);

      // soma créditos por volume
      volumeCredits += Math.max(perf.audience - 30, 0);
      // soma um crédito extra para cada
      // dez espectadores de comédia
      if ("comedy" === playFor(perf).type) volumeCredits += Math.floor(perf.audience / 5);
      // exibe a linha para esta requisição
      result += ` ${playFor(perf).name}: ${format(thisAmount/100)} (${perf.audience} seats)\n`;
      totalAmount += thisAmount;
    }
    result += `Amount owed is ${format(totalAmount/100)}\n`;
    result += `You earned ${volumeCredits} credits\n`;
    return result;
```

function statement…

```
  function amountFor(aPerformance~~, play~~) {
    let result = 0;
```

```
  switch (playFor(aPerformance).type) {
  case "tragedy":
    result = 40000;
    if (aPerformance.audience > 30) {
      result += 1000 * (aPerformance.audience - 30);
    }
    break;
  case "comedy":
    result = 30000;
    if (aPerformance.audience > 20) {
      result += 10000 + 500 * (aPerformance.audience - 20);
    }
    result += 300 * aPerformance.audience;
    break;
  default:
      throw new Error(`unknown type: ${playFor(aPerformance).type}`);
  }
  return result;
}
```

E compilo-testo-faço commit novamente.

Essa refatoração deixa alguns programadores alarmados. Anteriormente, o código para procurar a peça era executado uma vez a cada iteração do laço; agora ele é executado três vezes. Falarei sobre a inter-relação entre refatoração e desempenho mais tarde; por enquanto, porém, direi apenas que é pouco provável que essa modificação afete significativamente o desempenho, e, mesmo que o fizesse, é muito mais fácil melhorar o desempenho de uma base de código bem fatorada.

A grande vantagem de remover variáveis locais é que isso facilita muito as extrações, pois haverá menos escopo local com o qual lidar. Na verdade, eu geralmente removo as variáveis locais antes de fazer qualquer extração.

Agora que já cuidei dos argumentos de amountFor, volto a observar o local em que essa função é chamada. Ela está sendo usada para definir uma variável temporária que não é atualizada novamente, portanto aplico *Internalizar variável (Inline Variable)*.

nível mais alto…

```
function statement (invoice, plays) {
  let totalAmount = 0;
  let volumeCredits = 0;
  let result = `Statement for ${invoice.customer}\n`;
  const format = new Intl.NumberFormat("en-US",
```

```
                        { style: "currency", currency: "USD",
                          minimumFractionDigits: 2 }).format;

  for (let perf of invoice.performances) {

    // soma créditos por volume
    volumeCredits += Math.max(perf.audience - 30, 0);
    // soma um crédito extra para cada dez espectadores de comédia
    if ("comedy" === playFor(perf).type) volumeCredits += Math.floor(perf.audience / 5);

    // exibe a linha para esta requisição
    result += ` ${playFor(perf).name}: ${format(amountFor(perf)/100)} (${perf.audience} seats)\n`;
    totalAmount += amountFor(perf);
  }
  result += `Amount owed is ${format(totalAmount/100)}\n`;
  result += `You earned ${volumeCredits} credits\n`;
  return result;
```

Extraindo créditos por volume

Eis o estado atual do corpo da função statement:

nível mais alto…

```
function statement (invoice, plays) {
  let totalAmount = 0;
  let volumeCredits = 0;
  let result = `Statement for ${invoice.customer}\n`;
  const format = new Intl.NumberFormat("en-US",
                        { style: "currency", currency: "USD",
                          minimumFractionDigits: 2 }).format;
  for (let perf of invoice.performances) {

    // soma créditos por volume
    volumeCredits += Math.max(perf.audience - 30, 0);
    // soma um crédito extra para cada
    // dez espectadores de comédia
    if ("comedy" === playFor(perf).type) volumeCredits += Math.floor(perf.audience / 5);

    // exibe a linha para esta requisição
    result += ` ${playFor(perf).name}: ${format(amountFor(perf)/100)} (${perf.audience} seats)\n`;
    totalAmount += amountFor(perf);
  }
  result += `Amount owed is ${format(totalAmount/100)}\n`;
  result += `You earned ${volumeCredits} credits\n`;
  return result;
```

Tenho agora a vantagem de ter removido a variável play, pois facilita extrair o

cálculo dos créditos por volume por causa da remoção de uma das variáveis com escopo local.

Ainda tenho de lidar com as outras duas variáveis. Novamente, `perf` é fácil de passar, porém `volumeCredits` é um pouco mais complicado, pois é um acumulador atualizado a cada passagem pelo laço. Assim, minha melhor aposta é inicializar uma sombra dela na função extraída e devolvê-la.

function statement…

```
function volumeCreditsFor(perf) {
  let volumeCredits = 0;
  volumeCredits += Math.max(perf.audience - 30, 0);
  if ("comedy" === playFor(perf).type) volumeCredits += Math.floor(perf.audience / 5);
  return volumeCredits;
}
```

nível mais alto…

```
function statement (invoice, plays) {
  let totalAmount = 0;
  let volumeCredits = 0;
  let result = `Statement for ${invoice.customer}\n`;
  const format = new Intl.NumberFormat("en-US",
                        { style: "currency", currency: "USD",
                          minimumFractionDigits: 2 }).format;
  for (let perf of invoice.performances) {
    volumeCredits += volumeCreditsFor(perf);
    // exibe a linha para esta requisição
    result += ` ${playFor(perf).name}:
        {format(amountFor(perf)/100)} (${perf.audience} seats)\n`;
    totalAmount += amountFor(perf);
  }
  result += `Amount owed is ${format(totalAmount/100)}\n`;
  result += `You earned ${volumeCredits} credits\n`;
  return result;
```

Removo o comentário desnecessário (e, nesse caso, certamente enganador).

Compilo-testo-faço commit desse código, e então renomeio as variáveis na nova função.

function statement…

```
function volumeCreditsFor(aPerformance) {
  let result = 0;
  result += Math.max(aPerformance.audience - 30, 0);
  if ("comedy" === playFor(aPerformance).type) result +=
```

```
    Math.floor(aPerformance.audience / 5);
  return result;
}
```

Mostrei tudo em um só passo, mas, como antes, renomeei as variáveis uma de cada vez, fazendo uma compilação-teste-commit a cada vez.

Removendo a variável format

Vamos observar o método principal statement novamente:

nível mais alto…

```
function statement (invoice, plays) {
  let totalAmount = 0;
  let volumeCredits = 0;
  let result = `Statement for ${invoice.customer}\n`;
  const format = new Intl.NumberFormat("en-US",
                        { style: "currency", currency: "USD",
                          minimumFractionDigits: 2 }).format;
  for (let perf of invoice.performances) {
    volumeCredits += volumeCreditsFor(perf);

    // exibe a linha para esta requisição
    result += ` ${playFor(perf).name}:
        ${format(amountFor(perf)/100)} (${perf.audience} seats)\n`;
    totalAmount += amountFor(perf);
  }
  result += `Amount owed is ${format(totalAmount/100)}\n`;
  result += `You earned ${volumeCredits} credits\n`;
  return result;
```

Conforme sugeri antes, variáveis temporárias podem ser um problema. Elas são úteis somente em sua própria rotina e, desse modo, incentivam rotinas longas e complexas. Meu próximo passo, então, é substituir algumas delas. A mais fácil é format. Esse é um caso de atribuição de uma função a uma variável temporária, a qual prefiro substituir por uma função declarada.

function statement…

```
function format(aNumber) {
  return new Intl.NumberFormat("en-US",
                      { style: "currency", currency: "USD",
                        minimumFractionDigits: 2 }).format(aNumber);
}
```

nível mais alto…

```
function statement (invoice, plays) {
  let totalAmount = 0;
  let volumeCredits = 0;
  let result = `Statement for ${invoice.customer}\n`;
  for (let perf of invoice.performances) {
    volumeCredits += volumeCreditsFor(perf);

    // exibe a linha para esta requisição
    result += ` ${playFor(perf).name}:
        ${format(amountFor(perf)/100)} (${perf.audience} seats)\n`;
    totalAmount += amountFor(perf);
  }
  result += `Amount owed is ${format(totalAmount/100)}\n`;
  result += `You earned ${volumeCredits} credits\n`;
  return result;
```

Embora alterar uma variável de função para uma função declarada seja uma refatoração, não nomeei nem incluí essa refatoração no catálogo. Há muitas refatorações que não achei que fossem suficientemente importantes a esse ponto. Essa é simples de fazer e relativamente rara, portanto não achei que valesse a pena incluí-la.

Não gosto do nome – "format" não comunica realmente o que ela faz. "formatAsUSD" seria um pouco longo demais, pois é usada em um template de string, particularmente nesse escopo pequeno. Acho que o fato de ela estar formatando um valor monetário é o que deve ser enfatizado nesse caso, portanto, escolhi um nome que sugere isso e apliquei *Mudar declaração de função (Change Function Declaration)*.

nível mais alto…

```
function statement (invoice, plays) {
  let totalAmount = 0;
  let volumeCredits = 0;
  let result = `Statement for ${invoice.customer}\n`;
  for (let perf of invoice.performances) {
    volumeCredits += volumeCreditsFor(perf);
    // exibe a linha para esta requisição
    result += ` ${playFor(perf).name}:
        ${usd(amountFor(perf))} (${perf.audience} seats)\n`;
    totalAmount += amountFor(perf);
  }
  result += `Amount owed is ${usd(totalAmount)}\n`;
  result += `You earned ${volumeCredits} credits\n`;
```

```
  return result;
```

function statement…

```
function usd(aNumber) {
  return new Intl.NumberFormat("en-US",
                        { style: "currency", currency: "USD",
                          minimumFractionDigits: 2 }).format(aNumber/100);
}
```

Nomear é importante, mas é também complicado. Dividir uma função grande em funções menores só trará vantagens se os nomes forem bons. Com nomes bons, não preciso ler o corpo da função para ver o que ela faz. Entretanto, é difícil criar nomes apropriados na primeira vez, portanto uso o melhor nome em que puder pensar no momento, e não hesito em renomear mais tarde. Com frequência, uma segunda passagem pelo código é necessária para perceber qual é realmente o melhor nome.

Enquanto modifico o nome, também passo a divisão duplicada por 100 para dentro da função. Armazenar um valor monetário como centavos inteiros é uma abordagem comum – ela evita os perigos de armazenar valores monetários fracionários como números de ponto flutuante, ao mesmo tempo em que me permite usar operadores aritméticos. Sempre que eu quiser exibir esses números inteiros que representam centavos, porém, preciso de um valor decimal, portanto, minha função de formatação deve cuidar da divisão.

Removendo o total de créditos por volume

Minha próxima variável visada é volumeCredits. Esse é um caso mais intrincado, pois ela é calculada durante as iterações no laço. Meu primeiro passo, então, é usar *Dividir laço (Split Loop)* para separar a acumulação em volumeCredits.

nível mais alto…

```
function statement (invoice, plays) {
  let totalAmount = 0;
  let volumeCredits = 0;
  let result = `Statement for ${invoice.customer}\n`;

  for (let perf of invoice.performances) {

    // exibe a linha para esta requisição
    result += ` ${playFor(perf).name}:
        ${usd(amountFor(perf))} (${perf.audience} seats)\n`;
    totalAmount += amountFor(perf);
  }
```

```
  for (let perf of invoice.performances) {
    volumeCredits += volumeCreditsFor(perf);
  }

  result += `Amount owed is ${usd(totalAmount)}\n`;
  result += `You earned ${volumeCredits} credits\n`;
  return result;
```

Depois de fazer isso, posso usar *Deslocar instruções (Slide Statements)* para mover a declaração da variável para perto do laço.

nível mais alto…

```
function statement (invoice, plays) {
  let totalAmount = 0;
  let result = `Statement for ${invoice.customer}\n`;
  for (let perf of invoice.performances) {

    // exibe a linha para esta requisição
    result += ` ${playFor(perf).name}:
        ${usd(amountFor(perf))} (${perf.audience} seats)\n`;
    totalAmount += amountFor(perf);
  }
  let volumeCredits = 0;
  for (let perf of invoice.performances) {
    volumeCredits += volumeCreditsFor(perf);
  }
  result += `Amount owed is ${usd(totalAmount)}\n`;
  result += `You earned ${volumeCredits} credits\n`;
    return result;
```

Reunir tudo que atualiza a variável `volumeCredits` facilita o uso de *Substituir variável temporária por consulta (Replace Temp with Query)*. Como antes, o primeiro passo é aplicar *Extrair função (Extract Function)* ao cálculo geral da variável.

function statement…

```
function totalVolumeCredits() {
  let volumeCredits = 0;
  for (let perf of invoice.performances) {
    volumeCredits += volumeCreditsFor(perf);
  }
  return volumeCredits;
}
```

nível mais alto…

```
function statement (invoice, plays) {
  let totalAmount = 0;
  let result = `Statement for ${invoice.customer}\n`;
  for (let perf of invoice.performances) {

    // exibe a linha para esta requisição
    result += ` ${playFor(perf).name}:
        ${usd(amountFor(perf))} (${perf.audience} seats)\n`;
    totalAmount += amountFor(perf);
  }
  let volumeCredits = totalVolumeCredits();
  result += `Amount owed is ${usd(totalAmount)}\n`;
  result += `You earned ${volumeCredits} credits\n`;
  return result;
```

Depois que tudo tiver sido extraído, posso aplicar *Internalizar variável (Inline Variable)*:

nível mais alto…

```
function statement (invoice, plays) {
  let totalAmount = 0;
  let result = `Statement for ${invoice.customer}\n`;
  for (let perf of invoice.performances) {

    // exibe a linha para esta requisição
    result += ` ${playFor(perf).name}:
        ${usd(amountFor(perf))} (${perf.audience} seats)\n`;
    totalAmount += amountFor(perf);
  }

  result += `Amount owed is ${usd(totalAmount)}\n`;
  result += `You earned ${totalVolumeCredits()} credits\n`;
  return result;
```

Deixe-me fazer uma pequena pausa para falar sobre o que acabei de fazer nesse caso. Em primeiro lugar, sei que os leitores, mais uma vez, ficarão preocupados com o desempenho em razão dessa alteração, pois muitas pessoas se sentem inquietas quando repetem um laço. Na maioria das vezes, porém, executar um laço como esse outra vez tem um efeito desprezível no desempenho. Se você medisse o tempo de execução do código antes e depois dessa refatoração, provavelmente não perceberia nenhuma mudança significativa na velocidade – e, em geral, é isso que acontece. A maioria dos programadores, até mesmo aqueles que são experientes, avaliam mal o desempenho real do código. Muitas de nossas intuições são desmentidas por

compiladores inteligentes, técnicas modernas de caching e afins. O desempenho do software em geral depende somente de algumas partes do código, e mudanças em qualquer outro lugar não causam nenhuma diferença significativa.

Contudo, “em geral” não é o mesmo que “sempre”. Às vezes, uma refatoração terá uma implicação significativa no desempenho. Mesmo nesse caso, eu geralmente sigo em frente e faço a refatoração porque é muito mais fácil ajustar o desempenho de um código bem fatorado. Se eu introduzir um problema significativo de desempenho durante a refatoração, gastarei tempo ajustando depois o desempenho. Pode ser que isso me leve a desfazer alguma refatoração que eu tenha feito antes – porém, na maioria das vezes, por causa da refatoração, consigo aplicar uma melhoria mais eficaz para ajuste do desempenho. Acabo ficando com um código com mais clareza e rapidez.

Assim, meu conselho geral sobre o desempenho com a refatoração é este: na maioria das vezes, você deverá ignorá-lo. Se sua refatoração introduzir reduções no desempenho, termine antes de refatorar e faça os ajustes de desempenho depois.

O segundo aspecto para o qual quero chamar a sua atenção diz respeito aos pequenos passos usados para remover `volumeCredits`. Eis os quatro passos, cada um seguido de compilação, testes e commit em meu repositório local de códigos-fontes:

- *Dividir laço (Split Loop)* para isolar a acumulação;
- *Deslocar instruções (Slide Statements)* para levar o código de inicialização para perto da acumulação;
- *Extrair função (Extract Function)* para criar uma função que calcula o total;
- *Internalizar variável (Inline Variable)* para remover totalmente a variável.

Confesso que nem sempre executo passos tão pequenos como esses – mas, sempre que a situação se torna difícil, minha primeira reação é executar passos menores. Em particular, caso um teste falhe durante uma refatoração, se eu não conseguir ver nem corrigir prontamente o problema, restauro o código para o meu último bom commit e refaço o que acabei de fazer em passos menores. Isso funciona porque faço commits com muita frequência e porque passos pequenos são o segredo para andar rapidamente, em particular quando trabalhamos com um código difícil.

Então repito essa sequência para remover `totalAmount`. Começo dividindo o

laço (compilo-testo-faço commit), depois desloco a inicialização da variável (compilo-testo-faço commit), e então extraio a função. Há um pequeno problema nesse caso: o melhor nome para a função é “totalAmount”, mas esse é o nome da variável, e não posso ter ambos ao mesmo tempo. Assim, dou um nome aleatório à nova função ao extraí-la (e compilo-testo-faço commit).

function statement…

```
function appleSauce() {
  let totalAmount = 0;
  for (let perf of invoice.performances) {
    totalAmount += amountFor(perf);
  }
  return totalAmount;
}
```

nível mais alto…

```
function statement (invoice, plays) {
  let result = `Statement for ${invoice.customer}\n`;
  for (let perf of invoice.performances) {
    result += ` ${playFor(perf).name}: ${usd(amountFor(perf))} (${perf.audience} seats)\n`;
  }
  let totalAmount = appleSauce();

  result += `Amount owed is ${usd(totalAmount)}\n`;
  result += `You earned ${totalVolumeCredits()} credits\n`;
  return result;
```

Então internalizo a variável (compilo-testo-faço commit) e renomeio a função para algo mais razoável (compilo-testo-faço commit).

nível mais alto…

```
function statement (invoice, plays) {
  let result = `Statement for ${invoice.customer}\n`;
  for (let perf of invoice.performances) {
    result += ` ${playFor(perf).name}: ${usd(amountFor(perf))} (${perf.audience} seats)\n`;
  }
  result += `Amount owed is ${usd(totalAmount())}\n`;
  result += `You earned ${totalVolumeCredits()} credits\n`;
  return result;
```

function statement…

```
function totalAmount() {
  let totalAmount = 0;
```

```
  for (let perf of invoice.performances) {
    totalAmount += amountFor(perf);
  }
  return totalAmount;
}
```

Também aproveito a oportunidade para modificar os nomes dentro das funções extraídas para que estejam de acordo com minha convenção.

function statement…

```
function totalAmount() {
  let result = 0;
  for (let perf of invoice.performances) {
    result += amountFor(perf);
  }
  return result;
}
function totalVolumeCredits() {
  let result = 0;
  for (let perf of invoice.performances) {
    result += volumeCreditsFor(perf);
  }
  return result;
}
```

Status: muitas funções aninhadas

Agora é uma boa hora para fazer uma pausa e observar o estado geral do código:

```
function statement (invoice, plays) {
  let result = `Statement for ${invoice.customer}\n`;
  for (let perf of invoice.performances) {
    result += ` ${playFor(perf).name}:
        ${usd(amountFor(perf))} (${perf.audience} seats)\n`;
  }
  result += `Amount owed is ${usd(totalAmount())}\n`;
  result += `You earned ${totalVolumeCredits()} credits\n`;
  return result;

  function totalAmount() {
    let result = 0;
    for (let perf of invoice.performances) {
      result += amountFor(perf);
    }
```

```
  return result;
}

function totalVolumeCredits() {
  let result = 0;
  for (let perf of invoice.performances) {
    result += volumeCreditsFor(perf);
  }
  return result;
}
function usd(aNumber) {
  return new Intl.NumberFormat("en-US",
                         { style: "currency", currency: "USD",
                           minimumFractionDigits: 2 }).format(aNumber/100);
}
function volumeCreditsFor(aPerformance) {
  let result = 0;
  result += Math.max(aPerformance.audience - 30, 0);
  if ("comedy" === playFor(aPerformance).type) result +=
      Math.floor(aPerformance.audience / 5);
  return result;
}
function playFor(aPerformance) {
  return plays[aPerformance.playID];
}
function amountFor(aPerformance) {
  let result = 0;
  switch (playFor(aPerformance).type) {
  case "tragedy":
    result = 40000;
    if (aPerformance.audience > 30) {
      result += 1000 * (aPerformance.audience - 30);
    }
    break;
  case "comedy":
    result = 30000;
    if (aPerformance.audience > 20) {
      result += 10000 + 500 * (aPerformance.audience - 20);
    }
    result += 300 * aPerformance.audience;
    break;
  default:
      throw new Error(`unknown type: ${playFor(aPerformance).type}`);
  }
```

```
    return result;
  }
}
```

A estrutura do código está muito melhor agora. A função de mais alto nível statement tem apenas sete linhas de código e tudo que ela faz é organizar a exibição do demonstrativo. Toda a lógica de cálculo foi transferida para uma porção de funções auxiliares. Isso facilita compreender cada cálculo individual bem como o fluxo geral do relatório.

Separando as fases de cálculo e de formatação

Até agora, minha refatoração teve como foco o acréscimo de estrutura suficiente à função para que eu pudesse entendê-la e vê-la em termos de suas partes lógicas. Em geral, é isso que ocorre no início da refatoração. Separar porções complicadas em partes menores é importante, assim como lhes dar bons nomes. Agora posso começar a me concentrar mais na mudança de funcionalidade que quero fazer – especificamente, oferecer uma versão HTML desse demonstrativo. Em várias aspectos, agora será muito mais fácil fazer isso. Com todo o código de cálculos separado, tudo que devo fazer é escrever uma versão HTML das sete linhas de código no início. O problema é que essas funções separadas estão aninhadas no método do demonstrativo textual, e eu não gostaria de copiar e colar esse código em uma nova função, apesar de ele estar bem organizado. Quero que as mesmas funções de cálculo sejam usadas pelas versões de texto e de HTML do demonstrativo.

Há várias maneiras de fazer isso, mas uma de minhas técnicas favoritas é *Separar em fases (Split Phase)*. Meu objetivo, nesse caso, é separar a lógica em duas partes: uma que calcule os dados necessários para o demonstrativo e outra que os renderize em texto ou em HTML. A primeira fase cria uma estrutura de dados intermediária que é passada para a segunda.

Começo um *Separar em fases (Split Phase)* aplicando *Extrair função (Extract Function)* ao código que compõe a segunda fase. Nesse caso, é o código de apresentação do demonstrativo, que, na verdade, é todo o conteúdo de statement. Isso, em conjunto com todas as função aninhadas, será colocado em uma função de nível mais alto própria que chamarei de renderPlainText.

```
function statement (invoice, plays) {
  return renderPlainText(invoice, plays);
}
function renderPlainText(invoice, plays) {
```

```
  let result = `Statement for ${invoice.customer}\n`;
  for (let perf of invoice.performances) {
    result += ` ${playFor(perf).name}: ${usd(amountFor(perf))} (${perf.audience} seats)\n`;
  }
  result += `Amount owed is ${usd(totalAmount())}\n`;
  result += `You earned ${totalVolumeCredits()} credits\n`;
  return result;
  function totalAmount() {...}
  function totalVolumeCredits() {...}
  function usd(aNumber) {...}
  function volumeCreditsFor(aPerformance) {...}
  function playFor(aPerformance) {...}
  function amountFor(aPerformance) {...}
```

Faço meu processo usual de compilar-testar-fazer commit, e então crio um objeto que atuará como minha estrutura de dados intermediária entre as duas fases. Passo esse objeto de dados como argumento para renderPlainText (compilo-testo-faço commit).

```
function statement (invoice, plays) {
  const statementData = {};
  return renderPlainText(statementData, invoice, plays);
}
function renderPlainText(data, invoice, plays) {
  let result = `Statement for ${invoice.customer}\n`;
  for (let perf of invoice.performances) {
    result += ` ${playFor(perf).name}: ${usd(amountFor(perf))} (${perf.audience} seats)\n`;
  }
  result += `Amount owed is ${usd(totalAmount())}\n`;
  result += `You earned ${totalVolumeCredits()} credits\n`;
  return result;
  function totalAmount() {...}
  function totalVolumeCredits() {...}
  function usd(aNumber) {...}
  function volumeCreditsFor(aPerformance) {...}
  function playFor(aPerformance) {...}
  function amountFor(aPerformance) {...}
```

Analiso agora os outros argumentos usados por renderPlainText. Quero passar os dados provenientes desses argumentos para a estrutura de dados intermediária, de modo que todo o código de cálculos passe para a função statement e renderPlainText atue exclusivamente nos dados passados para ela por meio do parâmetro data.

Meu primeiro passo é obter o cliente e adicioná-lo no objeto intermediário

(compilo-testo-faço commit).

```
function statement (invoice, plays) {
  const statementData = {};
  statementData.customer = invoice.customer;
  return renderPlainText(statementData, invoice, plays);
}
function renderPlainText(data, invoice, plays) {
  let result = `Statement for ${data.customer}\n`;
  for (let perf of invoice.performances) {
    result += ` ${playFor(perf).name}: ${usd(amountFor(perf))} (${perf.audience} seats)\n`;
  }
  result += `Amount owed is ${usd(totalAmount())}\n`;
  result += `You earned ${totalVolumeCredits()} credits\n`;
  return result;
```

De modo semelhante, acrescento as apresentações, e isso me permite remover o parâmetro invoice de renderPlainText (compilo-testo-faço commit).

nível mais alto…

```
function statement (invoice, plays) {
  const statementData = {};
  statementData.customer = invoice.customer;
  statementData.performances = invoice.performances;
  return renderPlainText(statementData, ~~invoice,~~ plays);
}
function renderPlainText(data, plays) {
  let result = `Statement for ${data.customer}\n`;
  for (let perf of data.performances) {
    result += ` ${playFor(perf).name}: ${usd(amountFor(perf))} (${perf.audience} seats)\n`;
  }
  result += `Amount owed is ${usd(totalAmount())}\n`;
  result += `You earned ${totalVolumeCredits()} credits\n`;
  return result;
```

function renderPlainText…

```
function totalAmount() {
  let result = 0;
  for (let perf of data.performances) {
    result += amountFor(perf);
  }
  return result;
}
function totalVolumeCredits() {
  let result = 0;
```

```
  for (let perf of data.performances) {
    result += volumeCreditsFor(perf);
  }
  return result;
}
```

Gostaria agora que o nome da peça fosse obtido dos dados intermediários. Para isso, tenho de enriquecer o registro das apresentações com os dados da peça (compilo-testo-faço commit).

```
function statement (invoice, plays) {
  const statementData = {};
  statementData.customer = invoice.customer;
  statementData.performances = invoice.performances.map(enrichPerformance);
  return renderPlainText(statementData, plays);

  function enrichPerformance(aPerformance) {
    const result = Object.assign({}, aPerformance);
    return result;
  }
```

No momento, estou simplesmente criando uma cópia do objeto de apresentação, mas, em breve, acrescentarei dados nesse novo registro. Uso uma cópia porque não quero modificar os dados passados para a função. Prefiro tratar os dados como imutáveis o máximo que puder – um estado mutável rapidamente se torna problemático.

O idiom `result = Object.assign({}, aPerformance)` parece muito estranho para pessoas que não tenham familiaridade com JavaScript. Ele faz uma cópia rasa (shallow copy). Eu preferiria ter uma função para isso, mas é um daqueles casos em que o idiom está tão entranhado no uso de JavaScript que escrever minha própria função pareceria fora de lugar para programadores JavaScript.

Agora que tenho um local para a peça, preciso adicioná-la. Para isso, tenho de aplicar *Mover função (Move Function)* em `playFor` e em `statement` (compilo-testo-faço commit).

function statement…

```
function enrichPerformance(aPerformance) {
  const result = Object.assign({}, aPerformance);
  result.play = playFor(result);
  return result;
}
```

```
function playFor(aPerformance) {
  return plays[aPerformance.playID];
}
```

Então substituo todas as referências a `playFor` em `renderPlainText` para usar o dado em seu lugar (compilo-testo-faço commit).

function renderPlainText…

```
  let result = `Statement for ${data.customer}\n`;
  for (let perf of data.performances) {
    result += ` ${perf.play.name}: ${usd(amountFor(perf))} (${perf.audience} seats)\n`;
  }
  result += `Amount owed is ${usd(totalAmount())}\n`;
  result += `You earned ${totalVolumeCredits()} credits\n`;
  return result;
function volumeCreditsFor(aPerformance) {
  let result = 0;
  result += Math.max(aPerformance.audience - 30, 0);
  if ("comedy" === aPerformance.play.type) result += Math.floor(aPerformance.audience / 5);
  return result;
}
function amountFor(aPerformance) {
  let result = 0;
  switch (aPerformance.play.type) {
  case "tragedy":
    result = 40000;
    if (aPerformance.audience > 30) {
      result += 1000 * (aPerformance.audience - 30);
    }
    break;
  case "comedy":
    result = 30000;
    if (aPerformance.audience > 20) {
      result += 10000 + 500 * (aPerformance.audience - 20);
    }
    result += 300 * aPerformance.audience;
    break;
  default:
      throw new Error(`unknown type: ${aPerformance.play.type}`);
  }
  return result;
}
```

Em seguida, movo `amountFor` de modo similar (compilo-testo-faço commit).

function statement…

```
function enrichPerformance(aPerformance) {
  const result = Object.assign({}, aPerformance);
  result.play = playFor(result);
  result.amount = amountFor(result);
  return result;
}
function amountFor(aPerformance) {...}
```

function renderPlainText…

```
  let result = `Statement for ${data.customer}\n`;
  for (let perf of data.performances) {
    result += ` ${perf.play.name}: ${usd(perf.amount)} (${perf.audience} seats)\n`;
  }
  result += `Amount owed is ${usd(totalAmount())}\n`;
  result += `You earned ${totalVolumeCredits()} credits\n`;
  return result;
function totalAmount() {
  let result = 0;
  for (let perf of data.performances) {
    result += perf.amount;
  }
  return result;
}
```

Na sequência, movo o cálculo dos créditos por volume (compilo-testo-faço commit).

function statement…

```
function enrichPerformance(aPerformance) {
  const result = Object.assign({}, aPerformance);
  result.play = playFor(result);
  result.amount = amountFor(result);
  result.volumeCredits = volumeCreditsFor(result);
  return result;
}
function volumeCreditsFor(aPerformance) {...}
```

function renderPlainText…

```
function totalVolumeCredits() {
  let result = 0;
  for (let perf of data.performances) {
    result += perf.volumeCredits;
  }
```

```
  return result;
}
```

Por fim, movo os dois cálculos de totais.

function statement…

```
const statementData = {};
statementData.customer = invoice.customer;
statementData.performances = invoice.performances.map(enrichPerformance);
statementData.totalAmount = totalAmount(statementData);
statementData.totalVolumeCredits = totalVolumeCredits(statementData);
return renderPlainText(statementData, plays);
function totalAmount(data) {...}
function totalVolumeCredits(data) {...}
```

function renderPlainText…

```
let result = `Statement for ${data.customer}\n`;
for (let perf of data.performances) {
  result += ` ${perf.play.name}: ${usd(perf.amount)} (${perf.audience} seats)\n`;
}
result += `Amount owed is ${usd(data.totalAmount)}\n`;
result += `You earned ${data.totalVolumeCredits} credits\n`;
return result;
```

Embora eu pudesse ter modificado os corpos dessas funções de totais de modo que usassem a variável `statementData` (pois ela está no escopo), prefiro passar o parâmetro explicitamente.

Depois de compilar-testar-fazer commit após a mudança, não consegui resistir a alguns usos rápidos de *Substituir laço por pipeline (Replace Loop with Pipeline)*.

function renderPlainText…

```
function totalAmount(data) {
  return data.performances
    .reduce((total, p) => total + p.amount, 0);
}
function totalVolumeCredits(data) {
  return data.performances
    .reduce((total, p) => total + p.volumeCredits, 0);
}
```

Agora extraio todo o código da primeira fase e o coloco em sua própria função (compilo-testo-faço commit).

nível mais alto…

```
function statement (invoice, plays) {
  return renderPlainText(createStatementData(invoice, plays));
}
function createStatementData(invoice, plays) {
  const statementData = {};
  statementData.customer = invoice.customer;
  statementData.performances = invoice.performances.map(enrichPerformance);
  statementData.totalAmount = totalAmount(statementData);
  statementData.totalVolumeCredits = totalVolumeCredits(statementData);
  return statementData;
```

Como o código está claramente separado agora, passo-o para o seu próprio arquivo (e altero o nome do resultado devolvido para que esteja de acordo com minha convenção usual).

statement.js…

```
import createStatementData from './createStatementData.js';
```

createStatementData.js…

```
export default function createStatementData(invoice, plays) {
  const result = {};
  result.customer = invoice.customer;
  result.performances = invoice.performances.map(enrichPerformance);
  result.totalAmount = totalAmount(result);
  result.totalVolumeCredits = totalVolumeCredits(result);
  return result;
function enrichPerformance(aPerformance) {...}
  function playFor(aPerformance) {...}
  function amountFor(aPerformance) {...}
  function volumeCreditsFor(aPerformance) {...}
  function totalAmount(data) {...}
  function totalVolumeCredits(data) {...}
```

Uma última execução de compilar-testar-fazer commit – e agora será fácil escrever uma versão HTML.

statement.js…

```
function htmlStatement (invoice, plays) {
  return renderHtml(createStatementData(invoice, plays));
}
function renderHtml (data) {
  let result = `<h1>Statement for ${data.customer}</h1>\n`;
  result += "<table>\n";
  result += "<tr><th>play</th><th>seats</th><th>cost</th></tr>";
  for (let perf of data.performances) {
```

```
    result += ` <tr><td>${perf.play.name}</td><td>${perf.audience}</td>`;
    result += `<td>${usd(perf.amount)}</td></tr>\n`;
  }
  result += "</table>\n";
  result += `<p>Amount owed is <em>${usd(data.totalAmount)}</em></p>\n`;
  result += `<p>You earned <em>${data.totalVolumeCredits}</em> credits</p>\n`;
  return result;
}
function usd(aNumber) {...}
```

(Passei usd para o nível mais alto para que renderHtml pudesse usá-lo.)

Status: separado em dois arquivos (e fases)

Este é um bom momento para fazer uma avaliação novamente e pensar no local em que está o código agora. Tenho dois arquivos de código.

statement.js

```
import createStatementData from './createStatementData.js';
function statement (invoice, plays) {
  return renderPlainText(createStatementData(invoice, plays));
}
function renderPlainText(data, plays) {
  let result = `Statement for ${data.customer}\n`;
  for (let perf of data.performances) {
    result += ` ${perf.play.name}: ${usd(perf.amount)} (${perf.audience} seats)\n`;
  }
  result += `Amount owed is ${usd(data.totalAmount)}\n`;
  result += `You earned ${data.totalVolumeCredits} credits\n`;
  return result;
}
function htmlStatement (invoice, plays) {
  return renderHtml(createStatementData(invoice, plays));
}
function renderHtml (data) {
  let result = `<h1>Statement for ${data.customer}</h1>\n`;
  result += "<table>\n";
  result += "<tr><th>play</th><th>seats</th><th>cost</th></tr>";
  for (let perf of data.performances) {
    result += ` <tr><td>${perf.play.name}</td><td>${perf.audience}</td>`;
    result += `<td>${usd(perf.amount)}</td></tr>\n`;
  }
  result += "</table>\n";
  result += `<p>Amount owed is <em>${usd(data.totalAmount)}</em></p>\n`;
```

```
  result += `<p>You earned <em>${data.totalVolumeCredits}</em> credits</p>\n`;
  return result;
}
function usd(aNumber) {
  return new Intl.NumberFormat("en-US",
                               { style: "currency", currency: "USD",
                                 minimumFractionDigits: 2 }).format(aNumber/100);
}
```

createStatementData.js

```
export default function createStatementData(invoice, plays) {
  const result = {};
  result.customer = invoice.customer;
  result.performances = invoice.performances.map(enrichPerformance);
  result.totalAmount = totalAmount(result);
  result.totalVolumeCredits = totalVolumeCredits(result);
  return result;

  function enrichPerformance(aPerformance) {
    const result = Object.assign({}, aPerformance);
    result.play = playFor(result);
    result.amount = amountFor(result);
    result.volumeCredits = volumeCreditsFor(result);
    return result;
  }
  function playFor(aPerformance) {
    return plays[aPerformance.playID]
  }
  function amountFor(aPerformance) {
    let result = 0;
    switch (aPerformance.play.type) {
    case "tragedy":
      result = 40000;
      if (aPerformance.audience > 30) {
        result += 1000 * (aPerformance.audience - 30);
      }
      break;
    case "comedy":
      result = 30000;
      if (aPerformance.audience > 20) {
        result += 10000 + 500 * (aPerformance.audience - 20);
      }
      result += 300 * aPerformance.audience;
      break;
```

```
    default:
        throw new Error(`unknown type: ${aPerformance.play.type}`);
    }
    return result;
}
function volumeCreditsFor(aPerformance) {
    let result = 0;
    result += Math.max(aPerformance.audience - 30, 0);
    if ("comedy" === aPerformance.play.type) result += Math.floor(aPerformance.audience / 5);
    return result;
}
function totalAmount(data) {
    return data.performances
        .reduce((total, p) => total + p.amount, 0);
}
function totalVolumeCredits(data) {
    return data.performances
        .reduce((total, p) => total + p.volumeCredits, 0);
}
```

Tenho mais código agora do que eu tinha no início: 70 linhas (sem contar htmlStatement), em comparação com 44, principalmente em razão do encapsulamento extra para deixar o código em funções. Se tudo mais for o mesmo, mais código será ruim – raramente, porém, tudo mais continuará igual. O código extra separa a lógica em partes identificáveis, isolando os cálculos dos demonstrativos de seu layout. Essa modularidade faz com que seja mais fácil para mim compreender as partes do código e como elas se encaixam. A concisão é a alma da perspicácia, mas a clareza é a alma de um software capaz de evoluir. Acrescentar essa modularidade me permite oferecer suporte ao código para a versão HTML sem qualquer duplicação nos cálculos.

Quando programar, siga a regra do acampamento: sempre deixe a base de código mais saudável do que estava quando você a encontrou.

Há outras modificações que eu poderia ter feito para simplificar a lógica de exibição, mas, por enquanto, o que fizemos bastará. Tenho sempre de encontrar um equilíbrio entre todas as refatorações que eu poderia fazer e o acréscimo de novas funcionalidades. Atualmente, a maioria das pessoas dá menos prioridade à refatoração – mas ainda há um equilíbrio. Minha regra é uma variação da regra do acampamento: sempre deixe a base de código mais saudável do que estava quando você a encontrou. Ela jamais será perfeita,

mas deverá ser melhor.

Reorganizando os cálculos por tipo

Voltarei minha atenção agora para a próxima mudança de funcionalidade: aceitar outras categorias de peças, cada uma com seus próprios cálculos de cobrança e de créditos por volume. No momento, para fazer alterações, tenho de entrar nas funções de cálculo e editar ali as condições. A função amountFor enfatiza o papel central que o tipo da peça desempenha na forma de fazer os cálculos – porém uma lógica condicional como essa tende a se enfraquecer à medida que novas modificações forem feitas, a menos que ela seja reforçada por elementos mais estruturais da linguagem de programação.

Há várias maneiras de introduzir uma estrutura para deixar isso explícito, mas, nesse caso, uma abordagem natural é o polimorfismo de tipo – um recurso de destaque na orientação a objetos clássica. A orientação a objetos clássica tem sido um aspecto controverso há muito tempo no mundo JavaScript, mas a versão ECMAScript 2015 oferece uma sintaxe e uma estrutura robustas para ela. Portanto, faz sentido usá-la em uma situação correta – como esta.

De modo geral, meu plano é definir uma hierarquia de herança com subclasses para comédia e tragédia que contenham a lógica de cálculo para esses casos. Quem fizer a chamada usará uma função polimórfica para o cálculo do valor, e a linguagem despachará para os diferentes cálculos associados às comédias e às tragédias. Criarei uma estrutura semelhante para o cálculo dos créditos por volume. Para isso, utilizarei duas refatorações. A refatoração principal é *Substituir condicional por polimorfismo (Replace Conditional with Polymorphism)*, que altera uma porção de código com condicionais por polimorfismo. Contudo, antes de usar *Substituir condicional por polimorfismo*, é necessário criar algum tipo de estrutura de herança. Tenho de criar uma classe que contenha as funções para calcular o valor cobrado e os créditos por volume.

Começo revendo o código dos cálculos. (Uma das consequências satisfatórias da refatoração anterior é que agora posso ignorar o código de formatação, desde que eu gere a mesma estrutura de dados de saída. Posso dar melhor suporte a isso acrescentando testes que verifiquem a estrutura de dados intermediária.)

createStatementData.js…

```
export default function createStatementData(invoice, plays) {
  const result = {};
  result.customer = invoice.customer;
  result.performances = invoice.performances.map(enrichPerformance);
  result.totalAmount = totalAmount(result);
  result.totalVolumeCredits = totalVolumeCredits(result);
  return result;

  function enrichPerformance(aPerformance) {
    const result = Object.assign({}, aPerformance);
    result.play = playFor(result);
    result.amount = amountFor(result);
    result.volumeCredits = volumeCreditsFor(result);
    return result;
  }
  function playFor(aPerformance) {
    return plays[aPerformance.playID]
  }
  function amountFor(aPerformance) {
    let result = 0;
    switch (aPerformance.play.type) {
    case "tragedy":
      result = 40000;
      if (aPerformance.audience > 30) {
        result += 1000 * (aPerformance.audience - 30);
      }
      break;
    case "comedy":
      result = 30000;
      if (aPerformance.audience > 20) {
        result += 10000 + 500 * (aPerformance.audience - 20);
      }
      result += 300 * aPerformance.audience;
      break;
    default:
        throw new Error(`unknown type: ${aPerformance.play.type}`);
    }
    return result;
  }
  function volumeCreditsFor(aPerformance) {
    let result = 0;
    result += Math.max(aPerformance.audience - 30, 0);
    if ("comedy" === aPerformance.play.type) result += Math.floor(aPerformance.audience / 5);
    return result;
```

```
}
function totalAmount(data) {
  return data.performances
    .reduce((total, p) => total + p.amount, 0);
}

function totalVolumeCredits(data) {
  return data.performances
    .reduce((total, p) => total + p.volumeCredits, 0);
}
```

Criando uma calculadora de apresentação

A função enrichPerformance é a chave, pois ela preenche a estrutura de dados intermediária com os dados de cada apresentação. No momento, ela chama funções com condicionais para o valor cobrado e os créditos por volume. O que preciso que ela faça é chamar essas funções em uma classe que as contenha. Como essa classe contém funções para calcular dados sobre as apresentações, vou chamá-la de calculadora de apresentação (performance calculator).

function createStatementData…

```
function enrichPerformance(aPerformance) {
  const calculator = new PerformanceCalculator(aPerformance);
  const result = Object.assign({}, aPerformance);
  result.play = playFor(result);
  result.amount = amountFor(result);
  result.volumeCredits = volumeCreditsFor(result);
  return result;
}
```

nível mais alto…

```
class PerformanceCalculator {
  constructor(aPerformance) {
    this.performance = aPerformance;
  }
}
```

Até agora, esse novo objeto não faz nada. Quero passar alguns comportamentos para ela – e gostaria de começar com o item mais simples de ser transferido, que é o registro da peça. Estritamente falando, não preciso fazer isso, pois a peça não varia polimorficamente; desse modo, porém, manterei todas as transformações de dados em um só local, e essa

consistência dará mais clareza ao código.

Para que isso funcione, usarei *Mudar declaração de função (Change Function Declaration)* a fim de passar a peça encenada para a calculadora.

function createStatementData…

```
function enrichPerformance(aPerformance) {
  const calculator = new PerformanceCalculator(aPerformance, playFor(aPerformance));
  const result = Object.assign({}, aPerformance);
  result.play = calculator.play;
  result.amount = amountFor(result);
  result.volumeCredits = volumeCreditsFor(result);
  return result;
}
```

class PerformanceCalculator…

```
class PerformanceCalculator {
  constructor(aPerformance, aPlay) {
    this.performance = aPerformance;
    this.play = aPlay;
  }
}
```

(Não estou mais dizendo para compilar-testar-fazer commit o tempo todo, pois suspeito que você esteja ficando cansado de ler isso. Porém, continuo executando esses passos em todas as oportunidades. Às vezes me canso de fazê-los – e dou uma chance aos erros de me morderem. Então aprendo minha lição e volto a entrar no ritmo.)

Passando funções para a calculadora

A próxima porção de lógica que moverei é muito maior e serve para calcular o valor de uma apresentação. Movi funções por aí casualmente enquanto reorganizava as funções aninhadas – mas esta é uma modificação mais profunda no contexto da função, portanto descreverei a refatoração *Mover função (Move Function)* passo a passo. A primeira parte dessa refatoração é copiar a lógica para o seu novo contexto – a classe de calculadora. Em seguida, adapto o código para se adequar à sua nova morada, alterando `aPerformance` para `this.performance` e `playFor(aPerformance)` para `this.play`.

class PerformanceCalculator…

```
get amount() {
  let result = 0;
```

```
    switch (this.play.type) {
      case "tragedy":
        result = 40000;
        if (this.performance.audience > 30) {
          result += 1000 * (this.performance.audience - 30);
        }
        break;
      case "comedy":
        result = 30000;
        if (this.performance.audience > 20) {
          result += 10000 + 500 * (this.performance.audience - 20);
        }
        result += 300 * this.performance.audience;
        break;
      default:
        throw new Error(`unknown type: ${this.play.type}`);
    }
    return result;
  }
```

Posso compilar neste ponto para verificar se há algum erro de compilação. A "compilação" em meu ambiente de desenvolvimento ocorre quando executo o código, então o que realmente faço é executar o Babel [babel]. Isso será suficiente para identificar qualquer erro de sintaxe na nova função – mas não muito mais que isso. Mesmo assim, esse pode ser um passo útil.

Depois que a nova função estiver adequada à sua nova morada, tomo a função original e a transformo em uma função de delegação que chamará a nova função.

function createStatementData…

```
  function amountFor(aPerformance) {
    return new PerformanceCalculator(aPerformance, playFor(aPerformance)).amount;
  }
```

Agora posso compilar-testar-fazer commit para garantir que o código esteja funcionando de forma apropriada em sua nova morada. Depois de fazer isso, utilizo *Internalizar função (Inline Function)* para chamar diretamente a nova função (compilo-testo-faço commit).

function createStatementData…

```
  function enrichPerformance(aPerformance) {
    const calculator = new PerformanceCalculator(aPerformance, playFor(aPerformance));
    const result = Object.assign({}, aPerformance);
```

```
  result.play = calculator.play;
  result.amount = calculator.amount;
  result.volumeCredits = volumeCreditsFor(result);
  return result;
}
```

Repito o mesmo processo para mover o cálculo dos créditos por volume.

function createStatementData…

```
function enrichPerformance(aPerformance) {
  const calculator = new PerformanceCalculator(aPerformance, playFor(aPerformance));
  const result = Object.assign({}, aPerformance);
  result.play = calculator.play;
  result.amount = calculator.amount;
  result.volumeCredits = calculator.volumeCredits;
  return result;
}
```

class PerformanceCalculator…

```
get volumeCredits() {
  let result = 0;
  result += Math.max(this.performance.audience - 30, 0);
  if ("comedy" === this.play.type) result += Math.floor(this.performance.audience / 5);
  return result;
}
```

Deixando a calculadora de apresentação polimórfica

Agora que tenho a lógica em uma classe, é hora de aplicar o polimorfismo. O primeiro passo é usar *Substituir código de tipos por subclasses (Replace Type Code with Subclasses)* para introduzir subclasses no lugar de código de tipos. Para isso, preciso criar subclasses da calculadora de apresentação e usar a subclasse apropriada em createPerformanceData. Para ter a subclasse correta, devo substituir a chamada do construtor por uma função, pois construtores JavaScript não podem devolver subclasses. Portanto, uso *"Substituir construtor por função de factory (Replace Constructor with Factory Function)"*.

function createStatementData…

```
function enrichPerformance(aPerformance) {
  const calculator = createPerformanceCalculator(aPerformance, playFor(aPerformance));
```

```
  const result = Object.assign({}, aPerformance);
  result.play = calculator.play;
  result.amount = calculator.amount;
  result.volumeCredits = calculator.volumeCredits;
  return result;
}
```

nível mais alto…

```
function createPerformanceCalculator(aPerformance, aPlay) {
  return new PerformanceCalculator(aPerformance, aPlay);
}
```

Com esse código sendo agora uma função, posso criar subclasses da calculadora de apresentação e fazer a função de criação selecionar qual delas será devolvida.

nível mais alto…

```
function createPerformanceCalculator(aPerformance, aPlay) {
    switch(aPlay.type) {
    case "tragedy": return new TragedyCalculator(aPerformance, aPlay);
    case "comedy" : return new ComedyCalculator(aPerformance, aPlay);
    default:
        throw new Error(`unknown type: ${aPlay.type}`);
    }
}
class TragedyCalculator extends PerformanceCalculator {
}
class ComedyCalculator extends PerformanceCalculator {
}
```

Esse código define a estrutura para o polimorfismo, de modo que posso agora passar para *Substituir condicional por polimorfismo (Replace Conditional with Polymorphism)*.

Começo pelo cálculo do valor para as tragédias.

class TragedyCalculator…

```
get amount() {
  let result = 40000;
  if (this.performance.audience > 30) {
    result += 1000 * (this.performance.audience - 30);
  }
  return result;
}
```

Somente o fato de ter esse método na subclasse é suficiente para sobrescrever

a condicional da superclasse. Todavia, se você for tão paranoico quanto eu, poderá fazer o seguinte:

class PerformanceCalculator…

```
get amount() {
  let result = 0;
  switch (this.play.type) {
    case "tragedy":
      throw 'bad thing';
    case "comedy":
      result = 30000;
      if (this.performance.audience > 20) {
        result += 10000 + 500 * (this.performance.audience - 20);
      }
      result += 300 * this.performance.audience;
      break;
    default:
      throw new Error(`unknown type: ${this.play.type}`);
  }
  return result;
}
```

Eu poderia ter removido o caso para tragédia e deixar que a ramificação default lançasse um erro. No entanto, gosto do lançamento explícito – e ele estará lá somente por mais alguns minutos (motivo pelo qual lancei uma string, e não um objeto de erro melhor).

Depois de compilar-testar-fazer commit desse código, passo para baixo também o caso da comédia.

class ComedyCalculator…

```
get amount() {
  let result = 30000;
  if (this.performance.audience > 20) {
    result += 10000 + 500 * (this.performance.audience - 20);
  }
  result += 300 * this.performance.audience;
  return result;
}
```

Posso agora remover o método `amount` da superclasse, pois ele jamais deverá ser chamado. Entretanto, será uma gentileza que faço a mim mesmo no futuro se eu deixar uma lápide.

class PerformanceCalculator…

```
get amount() {
  throw new Error('subclass responsibility');
}
```

A próxima condicional a ser substituída é o cálculo dos créditos por volume. Observando a discussão sobre futuras categorias de peças teatrais, percebo que a maioria das peças espera verificar se houve mais de 30 pessoas na audiência, com apenas algumas categorias introduzindo uma variação. Portanto, faz sentido deixar o caso mais comum na superclasse como default e deixar que as variações o sobrescrevam caso necessário. Assim, passei para baixo somente o caso das comédias:

class PerformanceCalculator…

```
get volumeCredits() {
  return Math.max(this.performance.audience - 30, 0);
}
```

class ComedyCalculator…

```
get volumeCredits() {
  return super.volumeCredits + Math.floor(this.performance.audience / 5);
}
```

Status: criando os dados com a calculadora polimórfica

É hora de refletir sobre o que a introdução da calculadora polimórfica fez com o código.

createStatementData.js

```
export default function createStatementData(invoice, plays) {
  const result = {};
  result.customer = invoice.customer;
  result.performances = invoice.performances.map(enrichPerformance);
  result.totalAmount = totalAmount(result);
  result.totalVolumeCredits = totalVolumeCredits(result);
  return result;

  function enrichPerformance(aPerformance) {
    const calculator = createPerformanceCalculator(aPerformance, playFor(aPerformance));
    const result = Object.assign({}, aPerformance);
    result.play = calculator.play;
```

```
    result.amount = calculator.amount;
    result.volumeCredits = calculator.volumeCredits;
    return result;
  }
  function playFor(aPerformance) {
    return plays[aPerformance.playID]
  }
  function totalAmount(data) {
    return data.performances
      .reduce((total, p) => total + p.amount, 0);
  }
  function totalVolumeCredits(data) {
    return data.performances
      .reduce((total, p) => total + p.volumeCredits, 0);
  }
}

function createPerformanceCalculator(aPerformance, aPlay) {
    switch(aPlay.type) {
    case "tragedy": return new TragedyCalculator(aPerformance, aPlay);
    case "comedy" : return new ComedyCalculator(aPerformance, aPlay);
    default:
        throw new Error(`unknown type: ${aPlay.type}`);
    }
}
class PerformanceCalculator {
  constructor(aPperformance, aPlay) {
    this.performance = aPerformance;
    this.play = aPlay;
  }
  get amount() {
    throw new Error('subclass responsibility');
  }
  get volumeCredits() {
    return Math.max(this.performance.audience - 30, 0);
  }
}
class TragedyCalculator extends PerformanceCalculator {
  get amount() {
    let result = 40000;
    if (this.performance.audience > 30) {
      result += 1000 * (this.performance.audience - 30);
    }
    return result;
```

```
  }
}
class ComedyCalculator extends PerformanceCalculator {
  get amount() {
    let result = 30000;
    if (this.performance.audience > 20) {
      result += 10000 + 500 * (this.performance.audience - 20);
    }
    result += 300 * this.performance.audience;
    return result;
  }
  get volumeCredits() {
    return super.volumeCredits + Math.floor(this.performance.audience / 5);
  }
}
```

Novamente, o tamanho do código aumentou, pois introduzi uma estrutura. A vantagem, nesse caso, é que os cálculos para cada tipo de peça estão agrupados. Se a maior parte das alterações forem nesse código, será conveniente tê-lo claramente separado dessa forma. Acrescentar um novo tipo de peça exige escrever uma nova subclasse e adicioná-la na função de criação.

O exemplo proporciona alguns insights sobre quando usar subclasses como essas será conveniente. Nesse caso, passei a verificação condicional de duas funções (amountFor e volumeCreditsFor) para uma única função construtora, createPerformanceCalculator. Quanto mais funções houver que dependam do mesmo tipo de polimorfismo, mais conveniente será essa abordagem.

Uma alternativa para o que foi feito nesse exemplo seria fazer createPerformanceData devolver a própria calculadora, em vez de a calculadora preencher a estrutura de dados intermediária. Uma das características interessantes do sistema de classes de JavaScript é que, com ele, usar getters é parecido com um acesso de dados comum. Minha opção entre devolver a instância ou calcular dados de saída separados depende de quem vai usar a estrutura de dados posteriormente. Nesse caso, preferi mostrar como usar a estrutura de dados intermediária para ocultar a decisão de usar uma calculadora polimórfica.

Considerações finais

Este é um exemplo simples, mas espero que ele dê a você uma noção de como é uma refatoração. Usei várias refatorações, incluindo *Extrair função*

(Extract Function), *Internalizar variável (Inline Variable)*, *Mover função (Move Function)* e *Substituir condicional por polimorfismo (Replace Conditional with Polymorphism)*.

Houve três etapas principais nesse episódio de refatoração: decomposição da função original em um conjunto de funções aninhadas, uso de *Separar em fases (Split Phase)* para separar o código de cálculos do código de exibição e, por fim, introdução de uma calculadora polimórfica para a lógica de cálculos. Cada uma delas acrescentou estrutura ao código, permitindo que eu comunicasse melhor o que o código estava fazendo.

Como ocorre com frequência na refatoração, as primeiras etapas foram direcionadas principalmente para tentar entender o que estava acontecendo. Eis uma sequência comum: ler o código, obter alguns insights e usar a refatoração para passar esse insight de sua mente de volta para o código. O código mais claro então facilita a sua compreensão, levando a insights mais profundos e a um ciclo de feedback positivo benéfico. Ainda há algumas melhorias que eu poderia ter feito, mas sinto que fiz o suficiente para passar no meu teste de deixar o código significativamente melhor do que estava quando o encontrei.

> *O verdadeiro teste para um bom código é a facilidade com que ele pode ser alterado.*

Estou falando de melhorar o código – mas os programadores adoram discutir sobre como é a aparência de um bom código. Sei que algumas pessoas têm objeção à minha preferência por funções pequenas e bem nomeadas. Se considerarmos que isso é uma questão de estética, em que nada é bom nem ruim, mas o pensamento o faz ser assim, não teremos nenhuma diretriz, exceto o gosto pessoal. Acredito, porém, que podemos ir além do gosto pessoal e dizer que o verdadeiro teste para um bom código é a facilidade com que podemos modificá-lo. O código deve ser óbvio: quando alguém tiver de fazer uma alteração, essa pessoa deverá localizar facilmente o código a ser modificado e fazer a alteração rapidamente, sem introduzir erros. Uma base de código saudável maximiza nossa produtividade, permitindo desenvolver mais funcionalidades para os nossos usuários, de modo mais rápido e mais barato. Para manter um código saudável, preste atenção no que está entre a equipe de programação e esse ideal, e então refatore para se aproximar do ideal.

Contudo, o mais importante a se aprender com esse exemplo é o ritmo da refatoração. Sempre que mostro às pessoas como faço uma refatoração, elas ficam surpresas com o tamanho pequeno de meus passos, em que cada passo deixa o código em um estado funcional no qual ele compila e os testes passam. Fiquei igualmente surpreso quando Kent Beck me mostrou como fazer isso em um quarto de hotel em Detroit duas décadas atrás. O segredo para uma refatoração eficaz é reconhecer que você será mais rápido se der passos minúsculos, pois o código nunca apresenta falhas e você pode combinar esses passos pequenos em alterações substanciais. Lembre-se disso – e o resto é silêncio.

CAPÍTULO 2

Princípios da refatoração

O exemplo no capítulo anterior deve ter dado a você uma boa noção do que é a refatoração. Agora que tem essa noção, é uma boa hora de dar um passo para trás e discutir alguns dos princípios gerais da refatoração.

Definindo a refatoração

Como muitos termos em desenvolvimento de software, o termo "refatoração" com frequência é usado de forma bastante flexível pelos seus adeptos. Eu uso o termo de modo mais preciso, e acho conveniente usá-lo dessa forma. (As definições a seguir são as mesmas que apresentei na primeira edição deste livro.) O termo "refatoração" pode ser usado como um substantivo, ou podemos usar o verbo "refatorar"[1]. Eis a definição do substantivo:

> Refatoração *(substantivo): uma modificação feita na estrutura interna do software para deixá-lo mais fácil de compreender e menos custoso para alterar, sem que seu comportamento observável seja alterado.*

Essa definição corresponde às refatorações nomeadas que mencionei nos exemplos anteriores, por exemplo, *Extrair função (Extract Function)* e *Substituir condicional por polimorfismo (Replace Conditional with Polymorphism)*.

Eis a definição do verbo:

> Refatorar *(verbo): reestruturar um software por meio da aplicação de uma série de refatorações, sem alterar o seu comportamento observável.*

Assim, eu poderia passar algumas horas refatorando e, nesse período, aplicar algumas dezenas de refatorações individuais.

Ao longo dos anos, muitas pessoas no mercado passaram a usar "refatoração" para se referir a qualquer tipo de limpeza no código – as definições anteriores, porém, apontam para uma abordagem específica de limpeza do código. A refatoração está totalmente ligada à aplicação de

pequenos passos que preservam o comportamento, efetuando uma grande mudança por meio do encadeamento de uma sequência desses passos. Cada refatoração individual é, por si só, bem pequena, ou uma combinação de passos pequenos. Como resultado, quando estou refatorando, meu código não permanece muito tempo em um estado com falhas, permitindo que eu pare a qualquer momento, mesmo que ainda não tenha terminado o trabalho.

> *Se alguém disser que seu código apresentava falhas durante alguns dias enquanto estava refatorando, pode ter certeza de que essa pessoa não estava refatorando.*

Uso "reestruturação" como um termo genérico para me referir a qualquer tipo de reorganização ou de limpeza de uma base de código, e vejo a refatoração como um tipo particular de reestruturação. A refatoração pode parecer ineficiente para as pessoas que deparam com ela pela primeira vez e me veem executando uma série de passos minúsculos, quando um único passo maior seria suficiente. Contudo, os passos minúsculos me permitem avançar mais rápido, pois podem ser combinados de modo bastante apropriado – e, essencialmente, porque não gasto nenhum tempo com depuração.

Em minhas definições, uso a expressão "comportamento observável". É um termo deliberadamente flexível, indicando que o código, de modo geral, deve fazer exatamente o mesmo que fazia antes de eu começar. Não significa que ele funcionará exatamente do mesmo modo – por exemplo, *Extrair função (Extract Function)* alterará a pilha de chamadas, portanto as características de desempenho poderão mudar –, mas nada com que o usuário deva se importar será alterado. Em particular, as interfaces para os módulos muitas vezes mudam como consequência de refatorações como *Mudar declaração de função (Change Function Declaration)* e *Mover função (Move Function)*. Qualquer bug que eu perceber durante uma refatoração deverá continuar presente depois dela (embora eu possa corrigir bugs latentes que ninguém tivesse percebido ainda).

A refatoração é muito semelhante à otimização de desempenho, pois ambas envolvem fazer manipulações no código, as quais não modificam a funcionalidade geral do programa. A diferença está no propósito: a refatoração é sempre feita para deixar o código "mais fácil de compreender e menos custoso para alterar". Isso pode deixar o código mais rápido ou mais

lento. Na otimização de desempenho, preocupo-me somente em deixar o código mais rápido, e fico preparado para ter um código final mais difícil de trabalhar caso eu realmente precise dessa melhoria no desempenho.

Dois chapéus

Kent Beck criou a metáfora dos dois chapéus. Quando uso refatoração para desenvolver software, divido meu tempo em duas atividades distintas: acrescentar funcionalidades e refatorar. Quando acrescento uma funcionalidade, não devo alterar um código existente; estou simplesmente adicionando novos recursos. Avalio o meu progresso acrescentando testes e fazendo com que eles passem. Quando refatoro, minha meta é não acrescentar funcionalidades – apenas reestruturo o código. Não acrescento nenhum teste (a menos que eu identifique um caso que não havia considerado antes); mudo os testes somente se tiver de acomodar uma alteração em uma interface.

À medida que desenvolvo um software, vejo-me frequentemente trocando de chapéu. Começo tentando adicionar uma nova funcionalidade, então percebo que isso seria muito mais fácil se o código estivesse estruturado de modo diferente. Assim, troco de chapéu e refatoro por um tempo. Depois que o código estiver mais bem estruturado, troco de chapéu novamente e acrescento a nova funcionalidade. Depois que a nova funcionalidade estiver pronta, percebo que criei o código dela de uma forma difícil de entender; assim, troco novamente de chapéu e refatoro. Tudo isso pode demorar apenas dez minutos, mas, durante esse período, sempre tenho consciência de qual chapéu estou usando e da sutil diferença que isso faz para o modo como programo.

Por que devemos refatorar?

Não quero defender que a refatoração seja a cura para todos os males de software. Ela não é uma solução mágica para todos os problemas. Apesar disso, é uma ferramenta importante – ajuda você a manter um bom controle sobre o código. A refatoração é uma ferramenta que pode – e deve – ser usada para diversos propósitos.

Refatoração melhora o design do software

Sem a refatoração, o design interno – a arquitetura – do software tende a

entrar em decadência. À medida que as pessoas alteram o código para atingir objetivos de curto prazo, muitas vezes sem uma total compreensão da arquitetura, o código perde a sua estrutura. Torna-se mais difícil para mim ver o design lendo o código. A perda de estrutura do código tem um efeito cumulativo. Quanto mais difícil ver o design do código, mais difícil será para mim preservá-lo, e mais rapidamente ele entrará em decadência. Refatorações regulares ajudam a manter o código em forma.

Um código com design ruim geralmente exige mais código para fazer as mesmas tarefas, em geral porque o código, literalmente, faz o mesmo em vários lugares. Assim, um aspecto importante para melhorar o design é eliminar códigos duplicados. Não é a redução na quantidade de código que fará o sistema executar mais rápido – o efeito no uso de memória pelos programas raramente será significativo. Reduzir a quantidade de código, porém, faz uma grande diferença na modificação do código. Quanto mais código houver, mais difícil será modificá-lo corretamente. Haverá mais código para que eu entenda. Altero essa parte do código aqui, mas o sistema não faz o que eu esperava porque não modifiquei aquela parte lá, que faz praticamente a mesma tarefa em um contexto um pouco diferente. Ao eliminar as duplicações, garanto que o código diga tudo uma só vez, e somente uma vez, e essa é a essência de um bom design.

Refatoração deixa o software mais fácil de entender

Em muitos aspectos, a programação é uma conversa com um computador. Escrevo um código que diz ao computador o que ele deve fazer, e ele responde fazendo exatamente o que eu lhe disser. Com o tempo, elimino a distância entre o que quero que ele faça e o que lhe digo que faça. A programação se refere a dizer exatamente o que quero. Todavia, é provável que haja outros usuários para o meu código-fonte. Em alguns meses, um ser humano tentará ler o meu código para fazer algumas alterações. Esse usuário, que com frequência esquecemos, na verdade é o mais importante. Quem se importa se o computador demorar alguns ciclos a mais para compilar algo? Entretanto, importará se um programador demorar uma semana para fazer uma alteração que teria consumido apenas uma hora se meu código fosse devidamente compreendido.

O problema é que, quando estou tentando fazer o programa funcionar, não

penso nesse futuro desenvolvedor. Fazer com que o código seja mais fácil de entender exige uma mudança no ritmo. A refatoração ajuda a tornar o meu código mais legível. Antes de refatorar, tenho um código que funciona, porém não está estruturado de forma ideal. Um pouco de tempo investido em refatoração pode fazer o código comunicar melhor o seu propósito – dizer mais claramente o que quero.

Não estou sendo necessariamente altruísta nesse caso. Muitas vezes, esse futuro desenvolvedor sou eu mesmo. Isso faz com que a refatoração seja mais importante ainda. Sou um programador muito preguiçoso. Uma das formas pelas quais minha preguiça se evidencia é o fato de eu nunca me lembrar de detalhes sobre o código que escrevo. Na verdade, tento não me lembrar deliberadamente de nada que eu possa consultar, pois tenho medo de que meu cérebro fique cheio. Faço questão de tentar colocar no código tudo que eu deveria recordar para que não precise me lembrar dessas informações. Desse modo, preocupo-me menos com a Maudite [maudite] matando as células de meu cérebro.

Refatoração me ajuda a encontrar bugs

Ajudar a entender o código também significa ajudar a localizar bugs. Admito que não sou extremamente bom em encontrar bugs. Algumas pessoas são capazes de ler uma porção de código e ver os bugs; eu não consigo. No entanto, percebo que, se eu refatorar o código, realizo um trabalho árduo para entender o que ele faz, e coloco essa nova compreensão diretamente de volta no código. Ao deixar a estrutura do programa mais clara, esclareço determinados pressupostos que fiz – até o ponto em que nem mesmo eu consigo evitar de localizar os bugs.

Isso me lembra uma afirmação que, muitas vezes, Kent Beck faz sobre si mesmo: “Não sou um ótimo programador; sou apenas um bom programador com ótimos hábitos”. A refatoração me ajuda a ser mais eficaz para escrever um código robusto.

Refatoração me ajuda a programar mais rapidamente

No final das contas, todos os pontos anteriores convergem para este: a refatoração me ajuda a desenvolver código mais rapidamente.

Parece contraintuitivo. Quando falo de refatoração, as pessoas podem ver

facilmente que ela promove uma melhoria de qualidade. Um design interno melhor, mais legibilidade, redução de bugs – tudo isso melhora a qualidade. Mas o tempo que gasto com a refatoração não deixa o desenvolvimento mais lento?

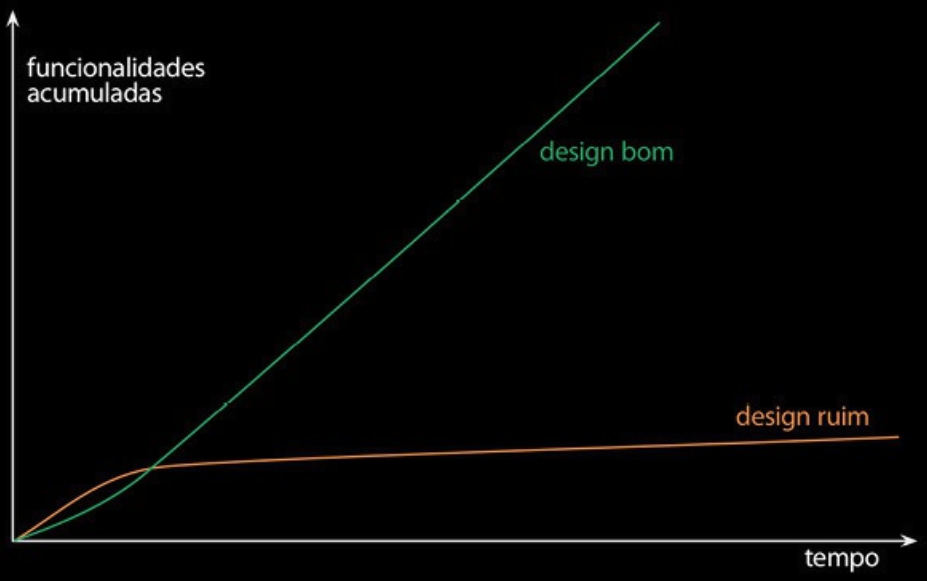
Quando converso com desenvolvedores de software que vêm trabalhando em um sistema há certo tempo, muitas vezes ouço eles dizendo que eram capazes de fazer progressos rapidamente no início, mas agora demoram muito mais para acrescentar novas funcionalidades. Cada funcionalidade nova exige mais e mais tempo para entender como ela se encaixará na base de código existente, e, depois de adicionada, com frequência surgem bugs que demoram mais ainda para serem corrigidos. A base de código começa a se parecer com uma série de remendos cobrindo remendos, e um exercício de arqueologia se faz necessário para descobrir como tudo funciona. Esse fardo faz com que o acréscimo de novas funcionalidades se torne lento – até o ponto de os desenvolvedores desejarem começar tudo de novo do zero.

Posso visualizar essa situação com o pseudográfico a seguir:

Algumas equipes, porém, descrevem uma experiência diferente. Elas percebem que conseguem acrescentar funcionalidades de modo *mais rápido* porque são capazes de tirar proveito do código existente, desenvolvendo mais rapidamente com base no que já está lá.

A diferença entre essas duas equipes está na qualidade interna do software. Um software com um bom design interno me permite saber facilmente como e onde devo fazer as alterações para acrescentar uma nova funcionalidade. Uma boa modularidade permite que eu entenda somente um pequeno subconjunto da base de código para fazer uma alteração. Se o código tiver clareza, haverá menos chances de eu introduzir um bug, e, se eu o fizer, o esforço de depuração será muito menor. Se estiver bem feita, minha base de código se transformará em uma plataforma para o desenvolvimento de novas

funcionalidades para o seu domínio.



Refiro-me a esse efeito como Hipótese da Estamina no Design (Design Stamina Hypothesis) [mf-dsh]: ao aplicar nossos esforços em um bom design interno, aumentamos a estamina do esforço de software, permitindo avançar mais rápido por mais tempo. Não posso provar que é isso que acontece, e é por isso que me refiro ao caso como uma hipótese. No entanto, ela explica a minha experiência, além da experiência de centenas de ótimos programadores que conheci ao longo de minha carreira.

Vinte anos atrás, de acordo com a sabedoria convencional, para ter esse tipo de design bom, ele deveria estar concluído antes que a programação começasse – porque, uma vez que escrevêssemos o código, só poderíamos encarar a sua decadência. A refatoração muda esse quadro. Sabemos agora que é possível melhorar o design de um código existente – desse modo, podemos construir e melhorar um design com o passar do tempo, mesmo que as necessidades do programa mudem. Como é muito difícil fazer um bom design com antecedência, a refatoração se torna essencial para chegar ao caminho virtuoso das funcionalidades rápidas.

Quando devemos refatorar?

Refatoração é algo que faço o tempo todo enquanto programo. Tenho percebido diversas formas pelas quais ela se encaixa em meu fluxo de trabalho.

A Regra dos Três

Eis uma orientação que Don Roberts me deu: na primeira vez que fizer algo, você simplesmente faz. Na segunda vez que fizer algo similar, você torce o nariz diante da duplicação, mas faz a duplicação, de qualquer modo. Na terceira vez que fizer algo similar, você refatora.

Ou, para aqueles que gostam de beisebol: três strikes, então você refatora.

Refatoração preparatória – facilitando o acréscimo de uma funcionalidade

A melhor hora para refatorar é logo antes de acrescentar uma nova funcionalidade na base de código. Quando faço isso, observo o código existente e muitas vezes percebo que, se ele estivesse estruturado de modo um pouco diferente, meu trabalho seria bem mais fácil. Talvez haja uma função que faça quase tudo de que preciso, porém tenha alguns valores literais que entrem em conflito com minhas necessidades. Sem uma refatoração, eu poderia copiar a função e alterar esses valores. No entanto, isso resultaria em um código duplicado – caso eu tenha de modificá-lo no futuro, terei de alterar dois lugares (e, pior de tudo, terei de encontrá-los). Além disso, copiar e colar não me ajudará se eu tiver de criar uma variante similar para uma nova funcionalidade no futuro. Então, usando meu chapéu de refatoração, utilizo *Parametrizar função (Parameterize Function)*. Depois disso, tudo que tenho a fazer é chamar a função com os parâmetros necessários.

> *É como se eu quisesse me deslocar 100 milhas (160,9 km) para leste, mas, em vez de ir devagar pela floresta, eu dirigisse 20 milhas (32,2 km) para o norte até a rodovia, e então seguisse 100 milhas para leste a uma velocidade três vezes maior do que seria possível caso eu tivesse avançado em linha reta. Quando as pessoas estiverem pressionando você para simplesmente seguir em linha reta, às vezes é necessário dizer: 'Espere, tenho que consultar o mapa e encontrar o caminho mais rápido'. A refatoração preparatória faz isso para mim.*
>
> *– Jessica Kerr*
> https://martinfowler.com/articles/preparatory-refactoring-example.html

O mesmo acontece quando corrigimos um bug. Depois de ter encontrado a

causa do problema, percebo que seria muito mais fácil corrigi-lo se eu unificasse as três porções de código copiado que estão causando o erro. Ou, quem sabe, separar uma lógica de atualização da lógica de consulta facilite evitar o emaranhado que está provocando o erro. Ao refatorar para melhorar a situação, também aumento as chances de que o bug permaneça corrigido e reduzirei as probabilidades de outros aparecerem nas mesmas reentrâncias do código.

Refatoração para compreensão: deixando o código mais fácil de entender

Antes de alterar um código, devo entender o que ele faz. Esse código pode ter sido escrito por mim ou por outra pessoa. Sempre que eu tiver de pensar para entender o que o código faz, pergunto a mim mesmo se posso refatorá-lo a fim de deixar essa compreensão mais aparente de forma imediata. Poderia estar olhando para uma lógica condicional estruturada de modo inconveniente. Talvez eu quisesse usar algumas funções existentes, mas gastei vários minutos para descobrir o que elas faziam porque seus nomes eram ruins.

Nesse ponto, tenho certa compreensão em minha mente, mas ela não é um registro muito bom para detalhes como esses. Como diz Ward Cunningham, ao refatorar, passo a compreensão que está em minha mente para o próprio código. Então testo essa compreensão executando o software para ver se ele continua funcionando. Se eu passar minha compreensão para o código, ela será preservada por mais tempo e estará visível aos meus colegas.

Isso não só me ajudará no futuro – com frequência, me ajuda nesse exato momento. Faço logo a refatoração para compreensão nos pequenos detalhes. Renomeio algumas variáveis, agora que entendo o que elas são, ou divido uma função longa em partes menores. Então, à medida que o código se torna mais claro, percebo que consigo enxergar detalhes do design que eu não conseguia ver antes. Se eu não tivesse modificado o código, provavelmente nunca teria visto esses detalhes, pois não sou inteligente o bastante para visualizar todas essas alterações em minha mente. Ralph Johnson descreve essas refatorações prévias como limpar a sujeira de uma janela para que você enxergue além. Quando estou estudando um código, a refatoração me leva a níveis mais elevados de compreensão que, de outra forma, eu não teria. Aqueles que menosprezam a refatoração para compreensão e a consideram

como uma manipulação inútil do código não percebem que, ao deixar de fazê-las, eles jamais verão as oportunidades ocultas por trás da confusão.

Refatoração para coleta de lixo

Uma variação da refatoração para compreensão é quando entendo o que o código faz, mas percebo que ele o faz de modo ruim. A lógica está desnecessariamente confusa, ou vejo funções que são quase idênticas e que poderiam ser substituídas por uma única função parametrizada. Há algumas contrapartidas nesse caso. Não gostaria de passar muito tempo com a atenção distante da tarefa que estou fazendo no momento, mas também não quero deixar lixo espalhado pelo código atrapalhando futuras alterações. Se for uma modificação simples, faço-a de imediato. Se exigir um pouco mais de esforço para corrigir, posso tomar nota dela e fazer a correção quando terminar minha tarefa imediata.

Às vezes, é claro, a correção exigirá algumas horas, e tenho tarefas mais urgentes para fazer. Mesmo nesse caso, porém, em geral vale a pena deixar o código um pouco melhor. Conforme diz o velho ditado sobre acampamentos, sempre deixe o local mais limpo do que estava quando você o encontrou. Se eu deixar o código um pouco melhor a cada vez que passar por ele, com o tempo ele estará corrigido. O aspecto interessante sobre a refatoração é que não deixo o código com falhas a cada pequeno passo – às vezes a tarefa poderá demorar meses para ser concluída, mas o código jamais terá falhas, mesmo que eu tenha feito somente parte do trabalho.

Refatorações planejadas e oportunistas

Os exemplos anteriores – refatorações preparatória, para compreensão e para coleta de lixo – são todos oportunistas. Não reservo tempo no início para gastar com a refatoração – em vez disso, faço a refatoração como parte do acréscimo de uma nova funcionalidade ou da correção de um bug. Faz parte de meu fluxo natural de programação. Independentemente de estar acrescentando uma funcionalidade ou corrigindo um bug, a refatoração me ajuda na tarefa imediata, além de preparar para deixar o trabalho mais fácil no futuro. Esse é um ponto importante, que frequentemente passa despercebido. A refatoração não é uma atividade separada da programação – não reservo mais tempo para ela do que reservo para escrever instruções `if`. Não reservo tempo em meus planejamentos para fazer refatorações: a maior parte delas

ocorre enquanto faço outras tarefas.

> *Você deve refatorar quando deparar com um código feio – mas um código excelente também precisa de muitas refatorações.*

Ver a refatoração como algo que as pessoas fazem para corrigir erros do passado ou limpar um código feio é, igualmente, um erro comum. Com certeza, você deve refatorar caso depare com um código feio, mas um código excelente também precisa de muitas refatorações. Sempre que escrevo um código, faço negociações – até que ponto preciso parametrizar, em que lugar devo traçar a linha entre as funções? As negociações que fiz corretamente para o conjunto de funcionalidades de ontem podem não ser mais as negociações corretas para as novas funcionalidades que estou acrescentando hoje. A vantagem é que um código claro será mais fácil de ser refatorado se eu precisar alterar essas negociações para que reflitam a nova realidade.

> *"Para cada mudança desejada, faça a mudança ser fácil (aviso: isso pode ser difícil), e então faça a fácil mudança."*
>
> *– Kent Beck:* https://twitter.com/kentbeck/status/250733358307500032

Por muito tempo, as pessoas pensavam na escrita de software como um processo de acréscimo: para adicionar novas funcionalidades, na maioria das vezes deveríamos acrescentar um novo código. Bons desenvolvedores, porém, sabem que, muitas vezes, o modo mais rápido de adicionar uma nova funcionalidade é alterar o código para facilitar a adição. Desse modo, o software jamais deveria ser pensado como algo "pronto". À medida que novas funcionalidades são necessárias, o software muda para refletir isso. Essas alterações, com frequência, podem ser maiores no código existente do que no código novo.

Tudo isso não significa que uma refatoração planejada seja sempre errada. Se uma equipe tiver negligenciado a refatoração, ela terá de dedicar tempo para deixar sua base de código em um estado melhor para receber novas funcionalidades, e uma semana investida em refatoração agora poderá compensar nos próximos meses. Às vezes, mesmo com refatorações constantes, posso ver uma área de problema crescer até o ponto em que seja necessário um esforço orquestrado para corrigi-la. Porém, episódios de refatoração planejada como esses devem ser raros. A maior parte do esforço de refatoração deve ser do tipo não planejado e oportunista.

Um pequeno conselho que ouvi é separar o trabalho de refatoração e de

acréscimos de novas funcionalidades em dois commits diferentes no sistema de controle de versões. A grande vantagem disso é o fato de eles poderem ser revisados e aprovados de forma independente. No entanto, não estou convencido disso. Com muita frequência, as refatorações estão intimamente entrelaçadas com o acréscimo de novas funcionalidades, e o tempo gasto para separá-los não vale a pena. Isso também poderia deixar a refatoração sem um contexto, dificultando justificar seus commits. Cada equipe deve fazer experimentos para descobrir o que funciona para elas; lembre-se apenas de que separar commits de refatorações não é um princípio evidente por si só – valerá a pena somente se facilitar a vida.

Refatoração de longo prazo

A maioria das refatorações pode ser feita em alguns minutos – no máximo, em algumas horas. No entanto, há esforços para algumas refatorações maiores que podem exigir semanas de uma equipe para terminar. Talvez seja necessário substituir uma biblioteca existente por uma nova. Ou extrair alguma seção de código e colocá-la em um componente que possa ser compartilhado com outra equipe. Ou corrigir alguma confusão terrível com dependências que eles permitiram que surgisse.

Mesmo em casos como esses, sou relutante quanto a ter uma equipe fazendo refatorações dedicadas. Com frequência, uma estratégia conveniente é concordar em trabalhar gradualmente no problema no decorrer das semanas seguintes. Sempre que qualquer pessoa se aproximar de algum código que esteja na zona de refatoração, ela se moverá um pouco na direção para a qual se quer melhorar. Isso tira proveito do fato de que a refatoração não causa falhas no código – cada pequena alteração ainda mantém tudo em um estado funcional. Para mudar de uma biblioteca para outra, comece introduzindo uma nova abstração que atue como uma interface para qualquer uma das bibliotecas. Depois que o código que faz as chamadas estiver usando essa abstração, será muito mais simples trocar uma biblioteca pela outra. (Essa tática se chama Branch por Abstração (Branch By Abstraction) [mf-bba].)

Refatorando em uma revisão de código

Algumas empresas fazem revisões de código regularmente; seria melhor se as demais também fizessem. Revisões de código ajudam a disseminar conhecimentos em uma equipe de desenvolvimento. As revisões ajudam os

desenvolvedores mais experientes a passar conhecimento para os menos experientes. Elas contribuem para que mais pessoas compreendam outros aspectos de um sistema de software de grande porte. Também são muito importantes para escrever um código claro. Meu código pode parecer claro para mim, mas não para a minha equipe. Isso é inevitável – é difícil para as pessoas se colocar no lugar de alguém que não tenha familiaridade com o que elas estão trabalhando. As revisões também dão oportunidades para mais pessoas sugerirem ideias úteis. Consigo pensar somente em algumas ideias boas em uma semana. Ter outras pessoas contribuindo facilita minha vida, portanto sempre procuro fazer revisões.

Tenho percebido que as refatorações me ajudam a revisar o código de outra pessoa. Antes de começar a usar refatorações, eu podia ler o código, compreendê-lo até certo ponto e fazer sugestões. Agora, quando tenho ideias, considero se elas podem ser facilmente implementadas aqui e ali com a refatoração. Em caso afirmativo, refatoro. Quando faço isso algumas vezes, consigo ver mais claramente como é a aparência do código com as sugestões implementadas. Não preciso imaginar como seria – sou capaz de ver. Como resultado, posso ter um segundo nível de ideias que eu jamais teria percebido caso não tivesse refatorado.

A refatoração também ajuda a obter resultados mais concretos da revisão de código. Não são somente sugestões; muitas delas são implementadas na própria ocasião. No final, você terminará o exercício com um sensação muito maior de realização.

O modo como incluirei a refatoração em uma revisão de código depende da natureza da revisão. O modelo comum de pull request, em que um revisor olha o código sem o autor original, não funciona muito bem. É melhor ter o autor original do código presente porque ele pode explicar o contexto para o código e apreciar de forma completa as intenções dos revisores quanto às mudanças. Minhas melhores experiências com isso se deram sentando-me lado a lado com o autor original, passando pelo código e refatorando à medida que fazíamos isso. A conclusão lógica desse estilo é a *programação aos pares* (pair programming): revisão de código contínua embutida no processo de programação.

O que devo dizer ao meu gerente?

Uma das perguntas mais comuns que me fazem é esta: “Como falar sobre refatoração a um gerente?”. Certamente já vi locais em que refatoração se

tornou um palavrão – com gerentes (e clientes) acreditando que a refatoração seja corrigir erros cometidos no passado ou um trabalho que não resulta em funcionalidades importantes. Isso é exacerbado por equipes que reservam semanas de pura refatoração no cronograma – em particular, se o que elas estiverem fazendo realmente não for refatoração, mas uma reestruturação menos cuidadosa, que cause falhas na base de código.

Para um gerente que seja verdadeiramente conhecedor de tecnologia e compreenda a hipótese da estamina no design, a refatoração não é difícil de justificar. Esses tipos de gerente devem incentivar a refatoração feita regularmente, e procurar sinais que mostrem que uma equipe não está fazendo o suficiente. Embora possa acontecer de as equipes exagerarem na refatoração, é muito mais raro que elas não a estejam fazendo suficientemente.

É claro que muitos gerentes e clientes não têm conhecimento técnico para saber como a saúde da base de código exerce impacto na produtividade. Para esses casos, ofereço meu conselho mais controverso: *não conte!*

Subversivo? Não acho. Desenvolvedores de software são profissionais. Nosso trabalho é construir um software eficaz o mais rápido que pudermos. De acordo com minha experiência, a refatoração contribui bastante para construir um software rapidamente. Se preciso acrescentar uma nova função e o design não está apropriado para a mudança, acho mais rápido refatorar antes, e então adicionar a função. Se preciso corrigir um bug, tenho de entender como o software funciona – e acho que a refatoração é o modo mais rápido de fazer isso. Um gerente orientado a cronogramas quer que eu faça minhas tarefas o mais rápido possível; o modo como faço é minha responsabilidade. Estou sendo pago pelo meu expertise em programar rapidamente de modo a ter novas funcionalidades, e a forma mais rápida é refatorando – portanto, refatoro.

Quando não devo refatorar?

Pode parecer que sempre recomendo refatorar – mas há casos em que não vale a pena.

Se eu deparar com um código confuso, mas que não precisa ser modificado, não terei de refatorá-lo. Alguns códigos feios que eu possa tratar como uma API podem permanecer feios. A refatoração me trará alguma vantagem somente se eu precisar entender como o código funciona.

Outro caso é aquele em que é mais fácil reescrever o código do que o

refatorar. Essa é uma decisão complicada. Com frequência, não sou capaz de dizer quão fácil é refatorar um código a menos que eu passe um tempo tentando e, desse modo, tenha uma ideia da dificuldade. A decisão de refatorar ou reescrever exige um bom julgamento e uma boa experiência, e não posso realmente a reduzir a um simples conselho único.

Problemas com a refatoração

Sempre que alguém defende alguma técnica, ferramenta ou arquitetura, procuro ver se há problemas. Poucas coisas na vida são somente dias ensolarados e céu azul. É necessário entender as contrapartidas para decidir quando e onde aplicar algo. Eu realmente acho que a refatoração é uma técnica valiosa – uma técnica que deveria ser mais usada pela maioria das equipes. Contudo, há problemas associados a ela, e é importante entender como eles se manifestam e como podemos reagir a eles.

Demorando mais para ter novas funcionalidades

Se você leu a seção anterior, já deverá saber a minha resposta. Embora muitas pessoas vejam o tempo gasto com refatoração como atraso para o desenvolvimento de novas funcionalidades, o propósito da refatoração como um todo é deixar as tarefas mais rápidas. Porém, embora isso seja verdade, também é verdade que a percepção sobre a refatoração causar atrasos ainda é comum – e talvez seja a principal barreira para as pessoas fazerem refatorações suficientes.

> *O principal propósito da refatoração é fazer com que programemos mais rápido, agregando mais valor com menos esforço.*

Há uma verdadeira negociação, nesse caso. Eu deparo com situações em que vejo uma refatoração (em larga escala) que realmente precisa ser feita, mas a nova funcionalidade que quero acrescentar é tão pequena que prefiro adicioná-la e não fazer a refatoração maior. É uma questão de avaliação – faz parte de minhas habilidades profissionais como programador. Não sou capaz de descrever facilmente, muito menos de quantificar, como faço essa negociação.

Tenho plena consciência de que uma refatoração preparatória em geral deixa uma alteração mais fácil; portanto, certamente irei fazê-la se perceber

que ela deixará minha nova funcionalidade mais fácil de implementar. Também me sinto mais inclinado a refatorar se esse for um problema que eu já tenha visto antes – às vezes, preciso ver um código particularmente feio umas duas vezes até decidir que devo refatorá-lo. Por outro lado, é menos provável que eu vá refatorar se o código estiver em uma parte com a qual raramente entro em contato e o custo da inconveniência não será sentido com muita frequência. Às vezes, adio uma refatoração porque não tenho certeza da melhoria que devo fazer, embora, em outras ocasiões, faço algo como um experimento para ver se a situação melhora.

Apesar disso, com base no que ouço meus colegas de mercado falarem, a evidência mostra que pouca refatoração é muito mais comum que refatoração demais. Em outras palavras, a maioria das pessoas deveria tentar refatorar com mais frequência. Talvez você tenha problemas em dizer qual é a diferença, em termos de produtividade, entre uma base de código saudável e uma base não saudável, em razão da falta de experiência suficiente com uma base de código saudável – da potencialidade de combinar facilmente as partes existentes e criar novas configurações a fim de obter novas funcionalidades complexas com rapidez.

Embora, em geral, os gerentes sejam criticados pelo hábito contraproducente de sacrificar a refatoração em nome da velocidade, já vi muitas vezes os próprios desenvolvedores fazerem isso. Às vezes, eles acham que não devem refatorar, mesmo que a liderança, na verdade, esteja a favor. Se você é líder técnico de uma equipe, é importante mostrar aos membros da equipe que você valoriza a melhoria da saúde de uma base de código. Essa capacidade de julgamento que mencionei antes, sobre refatorar ou não, exige anos de experiência para ser desenvolvida. Aqueles com menos experiência com refatoração precisam de muita orientação para acelerá-los nesse processo.

Acho, porém, que o modo mais perigoso, que faz as pessoas caírem em armadilhas, é quando elas tentam justificar a refatoração em termos de “código limpo”, “boas práticas de engenharia” ou motivos morais semelhantes. O ponto principal da refatoração não é mostrar como uma base de código é bonita, mas, sim, uma questão puramente econômica. Refatoramos porque isso nos torna mais rápidos – mais rápidos para acrescentar funcionalidades, mais rápidos para corrigir bugs. É importante ter isso em evidência na mente e na comunicação com outras pessoas. As vantagens econômicas da refatoração devem ser sempre o fator determinante,

e, quanto mais isso for compreendido pelos desenvolvedores, gerentes e clientes, mais da curva do “design bom” nós veremos.

Dono do código

Muitas refatorações envolvem fazer alterações que afetam não somente a parte interna de um módulo, mas seus relacionamentos com outras partes de um sistema. Se eu quiser renomear uma função, e puder encontrar todos os pontos em que ela é chamada, basta aplicar *Mudar declaração de função (Change Function Declaration)* e modificar a declaração e quem faz as chamadas de uma só vez. Às vezes, porém, essa refatoração simples não é possível. Talvez o código que faz as chamadas pertença a uma equipe diferente, e não tenho acesso de escrita em seu repositório. Pode ser que a função seja uma API declarada, usada pelos meus clientes – portanto, não sou sequer capaz de dizer se ela está sendo usada, muito menos por quem ou quanto. Essas funções fazem parte de uma interface publicada – uma interface usada por clientes de modo independente de quem a declara.

As fronteiras que determinam quem é o dono de um código atrapalham a refatoração porque não posso fazer os tipos de alterações que quero sem causar falhas aos meus clientes. Isso não impede a refatoração – ainda posso fazer muito –, mas impõe limitações. Quando renomeio uma função, tenho de usar *Renomear função (Rename Function)* e manter a declaração antiga como ponto de passagem para a nova função. Isso complica a interface – mas é o preço que devo pagar para evitar falhas em meus clientes. Posso marcar a interface antiga como obsoleta e, com o tempo, removê-la; às vezes, porém, terei de manter essa interface indefinidamente.

Em razão dessas complexidades, não recomendo uma definição extremamente rigorosa de quem é o dono do código. Algumas empresas gostam de que cada porção de código tenha um único programador como seu dono, e permitem que somente esse programador faça alterações nessa parte. Já vi uma equipe de três pessoas trabalhar de modo que cada um publicava interfaces para os outros dois. Isso gerava todo tipo de circularidades para preservar as interfaces, quando teria sido muito mais simples acessar a base de código e fazer as alterações. Minha preferência é permitir que uma equipe seja a dona do código – de modo que qualquer pessoa da mesma equipe possa modificar o código, mesmo que este tenha sido originalmente escrito por outra pessoa. Os programadores podem ter responsabilidades individuais por

áreas de um sistema, mas isso deve implicar que eles monitoram as alterações em suas áreas de responsabilidade, e não que eles as bloqueiem, por padrão.

Um esquema mais permissivo como esse para definir quem é o dono de um código pode existir até mesmo entre equipes. Algumas equipes incentivam um modelo do tipo código aberto, em que as pessoas de outras equipes podem modificar um branch (ramo) de seu código e enviar o commit para que seja aprovado. Isso permite que uma equipe modifique os clientes de suas funções – ela poderá apagar as declarações antigas depois que os commits para seus clientes tiverem sido aceitos. Muitas vezes, esse pode ser um bom compromisso entre uma definição rigorosa para o dono de um código e mudanças caóticas em sistemas de grande porte.

Branches

Atualmente – quando escrevi o livro –, uma abordagem comum nas equipes é que cada membro trabalhe em um branch da base de código usando um sistema de controle de versões, e faça um trabalho considerável nesse branch antes de integrá-lo com uma linha principal (mainline) – em geral, chamada de mestre (master) ou tronco (trunk) – compartilhada pela equipe. Com frequência, isso envolve implementar uma funcionalidade completa em um branch, sem integrá-la na linha principal, até que a funcionalidade esteja pronta para ser disponibilizada em um ambiente de produção. Fãs dessa abordagem argumentam que ela mantém a linha principal livre de qualquer código em processo de desenvolvimento, oferece um histórico de versões claro para os acréscimos de funcionalidades e permite que as funcionalidades sejam facilmente revertidas se causarem problemas.

Há desvantagens em branches de funcionalidade (feature branches) como esses. Quanto mais tempo eu trabalhar em um branch isolado, mais difícil será a tarefa de integrar o meu trabalho na linha principal quando eu terminar. A maioria das pessoas reduz essa dificuldade fazendo merges frequentes ou um rebase da linha principal para o seu próprio branch. Isso, contudo, não resolve realmente o problema se houver várias pessoas trabalhando em branches individuais de funcionalidades. Eu faço uma distinção entre merging e integração. Se faço merge da linha principal no meu código, esse é um movimento unidirecional – meu branch muda, mas a linha principal não. Uso “integrar” quando quero me referir a um processo bidirecional, que traz mudanças da linha principal para o meu branch, e o resultado é então enviado de volta para a linha principal, alterando ambos. Se Rachel está trabalhando

em seu branch, não veja as mudanças dela até que ela faça uma integração com a linha principal; nesse ponto, tenho de fazer um merge de suas alterações para o meu branch de funcionalidade, o que pode significar um trabalho considerável. A parte difícil desse trabalho é lidar com modificações semânticas. Sistemas modernos de controle de versões podem fazer maravilhas no merge de alterações complexas no texto do programa, mas são cegos à semântica do código. Se mudei o nome de uma função, minha ferramenta de controle de versões poderá facilmente integrar minhas modificações com as de Rachel. Porém, se, em seu branch, ela acrescentou uma chamada para uma função que renomeei em meu branch, o código falhará.

O problema de merges complicados piora exponencialmente à medida que o tamanho dos branches de funcionalidade aumentam. Integrar branches com quatro semanas de idade é duas vezes mais difícil que fazer a integração de branches com duas semanas de idade. Desse modo, muitas pessoas defendem a manutenção de branches de funcionalidade de curta duração – talvez com apenas cerca de dois dias. Outras pessoas, como eu, querem que eles sejam de duração mais curta ainda. Essa é uma abordagem chamada de CI (Continuous Integration, ou Integração Contínua), também conhecida como Trunk-Based Development (Desenvolvimento Baseado em Tronco). Com a CI, cada membro da equipe faz uma integração com a linha principal no mínimo uma vez por dia. Isso impede que os branches se afastem muito um do outro e, desse modo, reduz bastante a complexidade dos merges. A CI não vem de graça: significa que você usará práticas para garantir que a linha principal esteja saudável, aprenderá a dividir funcionalidades grandes em partes menores e usará toggles para funcionalidades (também conhecidos como flags de funcionalidade) para desativar qualquer funcionalidade em processo de desenvolvimento que não possa ser dividida.

Fãs de CI gostam dela parcialmente porque reduz a complexidade dos merges, mas o motivo principal para favorecer a CI é que ela é muito mais compatível com a refatoração. Com frequência, as refatorações envolvem fazer várias alterações pequenas por toda a base de código – o que é particularmente suscetível a conflitos semânticos nos merges (por exemplo, renomear uma função usada em vários lugares). Muitos de nós já viram equipes que trabalham com branches de funcionalidade acharem que as refatorações deixam os problemas de merge tão exacerbados que eles param com as refatorações. A CI e a refatoração funcionam bem juntos, e é por esse

motivo que Kent Beck as combinou na Extreme Programming (Programação Extrema).

Não estou dizendo que você jamais deva usar branches de funcionalidade. Se eles tiverem uma vida suficientemente curta, seus problemas serão bastante reduzidos. (Na verdade, usuários de CI em geral também usam branches, mas os integram diariamente com a linha principal.) Os branches de funcionalidade podem ser a técnica correta para projetos de código aberto, nos quais há commits não frequentes de programadores que você não conhece muito bem (e, portanto, em quem você não deposita confiança). Contudo, em uma equipe de desenvolvimento de tempo integral, o custo imposto pelos branches de funcionalidade na refatoração é excessivo. Mesmo que você não use totalmente a CI, sem dúvida incentivo você a integrar com a maior frequência possível. Considere também a evidência objetiva [Forsgren et al.] segundo a qual as equipes que usam CI são mais eficazes na entrega de softwares.

Testes

Uma das principais características da refatoração é o fato de ela não alterar o comportamento observável do programa. Se eu seguir cuidadosamente as refatorações, não deverei causar nenhuma falha – mas e se eu cometer um erro? (Ou, me conhecendo, não "se", mas, quando.) Erros acontecem, mas eles não são um problema, desde que eu os identifique rapidamente. Como cada refatoração é uma pequena alteração, se eu causar uma falha, terei apenas uma pequena alteração para observar a fim de encontrar essa falha – e se, apesar disso, eu não conseguir identificá-la, poderei restaurar a última versão em funcionamento, a partir de meu sistema de controle de versões.

O segredo, nesse caso, é ser capaz de identificar rapidamente um erro. Sendo realista, para isso, tenho de ser capaz de executar um conjunto abrangente de testes no código – e executá-lo rapidamente, de modo que eu não seja desencorajado a executá-lo com frequência. Isso significa que, na maioria dos casos, se eu quiser refatorar, terei de ter um código autotestável (self-testing code) [mf-stc].

Para alguns leitores, um código autotestável parece um requisito tão complicado que chega a não ser factível. Porém, nas duas últimas décadas, já vi muitas equipes desenvolverem software dessa maneira. Isso exige atenção e dedicação aos testes, mas as vantagens fazem com que realmente valha a pena. Um código autotestável não só permite refatoração, mas também faz

com que seja muito mais seguro acrescentar novas funcionalidades, pois posso localizar e eliminar rapidamente qualquer bug introduzido. O ponto principal, nesse caso, é que, quando um teste falha, posso olhar para a modificação que fiz entre a última vez que os testes estavam executando corretamente e o código atual. Com execuções frequentes dos testes, serão apenas algumas linhas de código. Por saber que foram essas linhas que causaram a falha, consigo encontrar o bug muito mais facilmente.

Isso também responde àqueles que estão preocupados com o fato de a refatoração trazer muito risco de introduzir bugs. Sem um código autotestável, essa seria uma preocupação razoável – e é por isso que coloco tanta ênfase em ter testes robustos.

Há outra maneira de lidar com o problema dos testes. Se eu usar um ambiente que tenha boas refatorações automatizadas, poderei confiar nessas refatorações mesmo sem executar testes. Posso então refatorar, desde que eu use somente as refatorações que estejam automatizadas de forma segura. Isso elimina muitas refatorações interessantes de meu cardápio, mas ainda me deixa o suficiente para ter alguns benefícios interessantes. Ainda prefiro ter um código autotestável, mas é uma opção conveniente para se ter no kit de ferramentas.

Essa abordagem também inspira um estilo de refatoração que use somente um conjunto limitado de refatorações comprovadamente seguras. Essas refatorações exigem que os passos sejam cuidadosamente seguidos, e são específicas para cada linguagem. Contudo, as equipes que as utilizam perceberam que podem fazer refatorações convenientes em bases de código grandes com uma cobertura de testes precária. Não enfocarei essa situação neste livro, pois é uma técnica mais recente, menos descrita e compreendida, a qual envolve uma atividade detalhada e específica para cada linguagem. (É, porém, um assunto sobre o qual espero falar mais em meu site no futuro. Para uma amostra sobre o assunto, consulte a descrição da Jay Bazuzi [Bazuzi] sobre uma forma mais segura de usar *Extrair método (Extract Method)* em C++.)

Não é nenhuma surpresa que um código autotestável esteja intimamente relacionado à CI (Continuous Integration, ou Integração Contínua) – é o mecanismo que usamos para identificar conflitos de integração semânticos. Práticas de teste como essa são outro componente da Extreme Programming, e uma parte essencial da Entrega Contínua (Continuous Delivery).

Código legado

A maioria das pessoas consideraria um grande legado como Algo Bom – mas esse é um dos casos em que a visão dos programadores é diferente. Um código legado em geral é complexo, vem frequentemente acompanhado de testes precários e, acima de tudo, foi escrito por Outra Pessoa (estremecimento).

A refatoração pode ser uma ferramenta incrível para ajudar a entender um sistema legado. Funções com nomes confusos podem ser renomeadas para que façam sentido, construções inconvenientes de programação podem ser reorganizadas e o programa pode passar de uma pedra bruta para uma joia polida. Porém, o dragão nesse conto feliz é a costumeira falta de testes. Se você tiver um sistema legado grande sem testes, não será possível refatorá-lo de forma segura para deixá-lo mais claro.

A resposta óbvia para esse problema é adicionar testes. Embora esse procedimento soe simples, apesar de trabalhoso, com frequência ele será muito mais intrincado na prática. Em geral, um sistema só será facilmente colocado em teste se tiver sido projetado com testes em mente – caso em que ele teria os testes e eu não estaria preocupado com isso.

Não há nenhuma forma simples de lidar com essa situação. O melhor conselho que posso dar é que você adquira uma cópia do livro *Trabalho eficaz com código legado* [Feathers] e siga as suas orientações. Não se preocupe com a idade do livro – seus conselhos continuam válidos, mesmo depois de uma década. Para resumir grosseiramente, o livro aconselha você a colocar o sistema em testes encontrando linhas de junção no programa nas quais você possa inserir os testes. Criar essas linhas de junção envolve refatoração – o que é muito mais perigoso, pois será feito sem testes, mas é um risco necessário para haver progressos. Essa é uma situação em que refatorações seguras e automatizadas podem ser uma bênção. Se tudo isso soar difícil, é porque é mesmo. Infelizmente, não há atalhos para sair de um buraco tão profundo como esse – e é por isso que sou um feroz defensor de escrever um código autotestável desde o início.

Mesmo quando tenho testes, não defendo tentar refatorar um sistema legado complicado e confuso para obter um código bonito de uma só vez. Prefiro lidar com ele em partes relevantes. Sempre que eu passar por uma seção de código, tento torná-la um pouco melhor – novamente, é como deixar uma área de acampamento mais limpa do que a encontrei. Se esse for um sistema

grande, farei mais refatorações em áreas que visito mais frequentemente – é a atitude correta a ser tomada porque, se eu precisar visitar um código com frequência, terei recompensas maiores por deixá-lo mais compreensível.

Bancos de dados

Quando escrevi a primeira edição deste livro, disse que refatorar bancos de dados era uma área problemática. Porém, um ano após a publicação do livro, isso deixou de ser verdade. Meu colega Pramod Sadalage desenvolveu uma abordagem para um design de banco de dados evolucionário [mf-evodb] e para refatoração de banco de dados [Ambler & Sadalage], o qual, nos dias de hoje, é amplamente usado. A essência da técnica consiste em combinar as mudanças estruturais no esquema de um banco de dados e o código de acesso com scripts de migração de dados que sejam facilmente usados em conjunto para lidar com alterações grandes.

Considere um exemplo simples em que um campo (coluna) é renomeado. Como em *Mudar declaração de função (Change Function Declaration)*, tenho de encontrar a declaração original da estrutura e todos que a usam e alterá-los em uma única mudança. A complicação, porém, é que também tenho de transformar qualquer dado que utilize o campo antigo para que passe a usar o novo campo. Escrevo uma pequena porção de código que faça essa transformação e a armazeno no sistema de controle de versões, junto com o código que altere qualquer estrutura declarada e as rotinas de acesso. Então, sempre que precisar fazer uma migração entre duas versões de banco de dados, executo todos os scripts de migração que existem entre minha cópia atual do banco de dados e a versão desejada.

Como no caso da refatoração usual, o segredo, nesse caso, é que cada alteração individual é pequena, mas captura uma modificação completa, de modo que o sistema continuará executando após a aplicação da migração. Manter as alterações pequenas significa que elas são fáceis de escrever, mas posso combinar várias delas em uma sequência, efetuando assim uma mudança significativa na estrutura do banco de dados e nos dados aí armazenados.

Uma diferença em comparação com as refatorações usuais é que as mudanças no banco de dados em geral estão mais bem separadas em várias versões para produção. Isso facilita reverter qualquer alteração que cause um problema no ambiente de produção. Assim, ao renomear um campo, meu primeiro commit seria acrescentar o novo campo no banco de dados, mas sem

usá-lo. Poderei então definir as atualizações de modo a atualizar tanto o campo antigo quanto o campo novo ao mesmo tempo. Posso então passar gradualmente as leituras para o novo campo. Somente depois que todos tiverem passado para o novo campo, e de eu ter esperado um tempo para que qualquer bug se evidencie, é que removo o campo antigo, não mais usado. Essa abordagem para mudanças em bancos de dados é um exemplo de uma abordagem genérica de mudança em paralelo [mf-pc] (também conhecida como expansão-contração).

Refatoração, arquitetura e Yagni

A refatoração mudou profundamente o modo como as pessoas pensam na arquitetura de software. No início de minha carreira, aprendi que design e arquitetura de software constituíam algo em que era necessário trabalhar, e principalmente concluir, antes que qualquer pessoa começasse a escrever algum código. Depois que o código estivesse escrito, sua arquitetura estava definida, e só poderia entrar em decadência por falta de cuidado.

A refatoração muda essa perspectiva. Ela me permite alterar significativamente a arquitetura do software em execução há anos em ambiente de produção. A refatoração é capaz de melhorar o design de um código existente, como implica o subtítulo deste livro. Porém, conforme mostrei antes, alterar um código 3 em geral é desafiador, particularmente quando não há testes razoáveis.

O verdadeiro impacto da refatoração na arquitetura está no modo como ela pode ser usada para compor uma base de código com um bom design, capaz de responder às necessidades de mudanças com elegância. O principal problema em concluir a arquitetura antes de programar é que uma abordagem como essa pressupõe que os requisitos de software podem ser compreendidos bem cedo. Todavia, a experiência mostra que essa é uma meta, com frequência, e até usualmente, inatingível. Inúmeras vezes, já vi pessoas compreenderem o que realmente precisavam que o software fizesse apenas depois que tiveram a chance de usá-lo, e vi o impacto que isso causava no trabalho delas.

Uma maneira de lidar com alterações futuras é inserir recursos que permitam ter flexibilidade no software. Quando escrevo uma função, posso ver que ela tem uma aplicabilidade geral. Para lidar com as diferentes circunstâncias nas quais é possível prever que ela será usada, posso vislumbrar uma dúzia de parâmetros que eu poderia acrescentar nessa função.

Esses parâmetros são recursos que permitem ter flexibilidade – e, como a maioria dos recursos, não são gratuitos. Acrescentar todos esses parâmetros complica a função para o caso único em que ela está sendo usada no momento. Se eu me esquecer de um parâmetro, toda a parametrização que acrescentei fará com que seja mais difícil adicionar outros. Muitas vezes percebo que meus recursos para flexibilidade estão incorretos – seja porque as necessidades de alterações não ocorreram do modo como eu esperava, seja porque o design de meu recurso apresentava falhas. Depois que levo tudo isso em consideração, na maioria das vezes, meus recursos de flexibilidade, na verdade, causam *lentidão* em minha capacidade de reagir a uma mudança.

Com a refatoração, posso usar uma estratégia diferente. Em vez de especular sobre a flexibilidade que precisarei ter no futuro e quais recursos permitirão fazer isso da melhor forma, implemento um software que resolva somente as necessidades compreendidas no momento, mas faço com que o design desse software seja excelente para essas necessidades. Conforme minha compreensão acerca das necessidades dos usuários mudar, utilizo a refatoração para adaptar a arquitetura de acordo com essas novas demandas. Posso incluir tranquilamente recursos que não aumentem a complexidade (por exemplo, funções pequenas, bem nomeadas), mas qualquer recurso de flexibilidade que complique o software deve mostrar que tem eficácia comprovada antes que eu o inclua. Se quem fizer as chamadas não usar valores diferentes para um parâmetro, não o acrescentarei na lista de parâmetros. Caso eu venha a ter de acrescentá-lo em algum momento, *Parametrizar função (Parameterize Function)* é uma refatoração fácil de ser aplicada. Em geral, acho conveniente estimar o nível de dificuldade para usar uma refatoração mais tarde a fim de dar suporte a uma mudança prevista. Apenas se eu perceber que seria significativamente mais difícil refatorar mais tarde é que considero acrescentar um recurso para prover flexibilidade agora.

Essa abordagem para o design recebe vários nomes: design simples, design incremental ou Yagni [mf-yagni] (originalmente, é um acrônimo para “You Aren’t Going to Need It”, isto é, “Você não vai precisar disso”). O Yagni não implica que você deixará de pensar na arquitetura, embora, às vezes, ele seja ingenuamente aplicado dessa forma. Penso no Yagni como um estilo diferente de incorporar a arquitetura e o design no processo de desenvolvimento – um estilo que não seria verossímil sem a base proporcionada pela refatoração.

Adotar o Yagni não significa que vou negligenciar todo o raciocínio prévio

sobre a arquitetura. Ainda há casos em que mudanças por refatoração são difíceis, e um raciocínio preparatório pode significar economia de tempo. Contudo, o ponto de equilíbrio foi bastante deslocado – estou muito mais inclinado a lidar com os problemas mais tarde, quando eu os tiver compreendido melhor. Tudo isso levou a uma crescente disciplina de arquitetura evolucionária [Ford et al.], em que os arquitetos exploram os padrões e as práticas que tirem proveito de nossa capacidade de iterar pelas decisões de arquitetura.

Refatoração e o processo mais amplo de desenvolvimento de software

Se você leu a seção anterior sobre problemas, uma lição que provavelmente extraiu dali é que a eficácia da refatoração está vinculada a outras práticas de software usadas por uma equipe. Com efeito, a adoção da refatoração logo no início faz parte da Extreme Programming [mf-xp] (XP), um processo que se tornou famoso por reunir um conjunto de práticas relativamente incomuns e interdependentes – como integração contínua, código autotestável e refatoração (os dois últimos se entrelaçaram dando origem ao desenvolvimento orientado a testes).

A Extreme Programming foi um dos primeiros métodos ágeis de software [mf-nm] e, por muitos anos, liderou a ascensão das técnicas ágeis. Há muitos projetos hoje em dia que usam métodos ágeis, a ponto de o Agile Thinking (Pensamento Ágil) ser considerado, de modo geral, uma técnica convencional – embora, na verdade, a maioria dos projetos “ágeis” só utilize o nome. Para realmente funcionar em modo ágil, uma equipe deve ser capacitada a fazer refatorações e ser entusiasmada com elas – e, para isso, muitos aspectos de seu processo devem estar alinhados com fazer das refatorações uma parte constante de seu trabalho.

A primeira base para a refatoração é ter um código autotestável (self-testing code). Com isso, quero dizer que deve haver uma suíte de testes automatizada, a qual posso executar e que me permita ter a confiança de que, caso eu cometa um erro em minha programação, algum teste falhará. Essa é uma base tão importante para a refatoração que dedicarei um capítulo para falar mais sobre o assunto.

Para refatorar em uma equipe, é importante que cada membro possa fazê-lo quando for necessário, sem interferir no trabalho dos demais. É por isso que

incentivo a Integração Contínua. Com a CI, os esforços de refatoração de cada membro são rapidamente compartilhados com seus colegas. Ninguém acabará fazendo novos trabalhos em interfaces que estão sendo removidas, e, se a refatoração causar algum problema no trabalho de outra pessoa, saberemos rapidamente. Um código autotestável também é um elemento essencial da Integração Contínua, portanto há uma forte sinergia entre as três práticas a saber: código autotestável, integração contínua e refatoração.

Com essas três práticas em ação, podemos usar a abordagem de design Yagni sobre a qual falei na seção anterior. A refatoração e o Yagni se reforçam entre si de forma positiva: a refatoração (e seus pré-requisitos) é uma base para o Yagni, e o Yagni facilita fazer refatorações. Isso ocorre porque é mais fácil modificar um sistema simples do que modificar um que tenha muitos recursos de flexibilidade especulativos incluídos. Encontre um equilíbrio entre essas práticas e você poderá entrar em um círculo virtuoso, com uma base de código que responda rapidamente às necessidades de mudanças e seja confiável.

Com essas práticas essenciais implantadas, temos a base para tirar proveito de outros elementos do modo de pensar ágil. A Entrega Contínua (Continuous Delivery) mantém nosso software em um estado em que é sempre possível lançar uma versão. É isso que permite que muitas empresas de web lancem atualizações várias vezes ao dia – mas, mesmo que não precisemos disso, o risco será reduzido e nos permitirá agendar nossos lançamentos de versões para satisfazer às necessidades do negócio, em vez de estarmos sujeitos às limitações tecnológicas. Com uma base técnica sólida, podemos reduzir drasticamente o tempo para fazer com que uma boa ideia esteja no código de produção, permitindo servir melhor aos nossos clientes. Além do mais, essas práticas aumentam a confiabilidade de nosso software, com menos bugs com os quais teríamos de gastar tempo para corrigir.

Falando dessa maneira, tudo parece muito simples – porém, não é assim na prática. O desenvolvimento de software, não importa a abordagem, é um negócio complicado, com interações complexas entre pessoas e máquinas. A abordagem que descrevo neste livro é uma forma comprovada de lidar com essa complexidade, mas, como qualquer abordagem, exige prática e habilidade.

Refatoração e desempenho

Uma preocupação comum na refatoração é o efeito que ela tem no

desempenho de um programa. Para deixar o software mais fácil de compreender, muitas vezes realizo alterações que farão o programa executar mais lentamente. Essa é uma questão importante. Não pertenço à escola de pensamento que ignora o desempenho em favor da pureza de design nem que vive na esperança de ter um hardware mais rápido. Softwares já foram rejeitados por serem lentos demais, e máquinas mais rápidas apenas fazem os objetivos se deslocarem para a frente. A refatoração certamente pode deixar um software mais lento – mas também pode fazer com que ele se torne mais favorável aos ajustes de desempenho. O segredo para um software rápido, exceto em contextos rígidos de tempo real, é escrever softwares ajustáveis primeiro, e depois os ajustar para velocidades que sejam suficientes.

Já vi três abordagens genéricas para escrever um software rápido. A mais séria delas é a de time budgeting (provisão de tempo), com frequência usada em sistemas de tempo real rígidos. À medida que você decompõe o design, cada componente recebe uma provisão (budget) para recursos – tempo e memória. Esse componente não deve exceder a sua provisão, embora um mecanismo para trocar recursos reservados seja permitido. O time budgeting foca a atenção em tempos de desempenho rígidos. É essencial para sistemas como marcapassos para o coração, em que dados atrasados são sempre dados ruins. Essa técnica não é apropriada para outros tipos de sistemas, por exemplo, sistemas de informação corporativos, com os quais geralmente trabalho.

A segunda abordagem é a abordagem da atenção constante (constant attention). Nesse caso, todo programador, o tempo todo, faz o que for possível para manter um bom desempenho. É uma abordagem comum, intuitivamente atraente – mas não funciona muito bem. Alterações que melhoram o desempenho em geral tornam mais difícil trabalhar com o programa. Isso deixa o desenvolvimento mais lento. Seria um custo que valeria a pena pagar se o software resultante fosse mais rápido – em geral, porém, não é o caso. As melhorias no desempenho estão espalhadas por todo o programa; cada melhoria é feita com uma perspectiva limitada acerca do comportamento do programa e, frequentemente, com uma compreensão errônea sobre como um compilador, um runtime e o hardware se comportam.

Demora um pouco para não criar nada

O processo de pagamento Comprehensive Compensation (Compensação Completa) da Chrysler estava executando de forma muito lenta. Embora

ainda estivéssemos na fase de desenvolvimento, o fato começou a nos incomodar porque os testes estavam ficando lentos.

Kent Beck, Martin Fowler e eu decidimos que tínhamos de corrigir isso. Enquanto eu esperava que nos reuníssemos, fiquei especulando, com base em meu amplo conhecimento do sistema, a respeito do motivo provável para a causa da lentidão. Pensei em diversas possibilidades e conversei com as pessoas sobre as mudanças que provavelmente seriam necessárias. Pensamos em algumas ideias realmente muito boas sobre o que faria o sistema executar mais rápido.

Então medimos o desempenho usando o profiler (gerador de perfil) de Kent. Nenhuma das possibilidades em que eu havia pensado tinha relação com o problema. Em vez disso, descobrimos que o sistema estava gastando metade do tempo criando instâncias de datas. Mais interessante ainda foi o fato de que todas as instâncias tinham a mesma dupla de valores.

Quando observamos a lógica de criação das datas, vimos algumas oportunidades para otimizar o modo como elas eram criadas. Todas as datas estavam passando por uma conversão de string, mesmo que nenhuma entrada externa estivesse envolvida. O código simplesmente usava conversão de string por ser conveniente para digitar. Talvez pudéssemos otimizar isso.

Em seguida, observamos como essas datas estavam sendo usadas. Percebemos que a maior parte delas criava instâncias de intervalos de datas, um objeto com uma data inicial e uma data final. Observando um pouco mais, percebemos que a maior parte desses intervalos de data estava vazia!

Enquanto trabalhávamos com intervalos de datas, usamos a convenção segundo a qual qualquer intervalo de data que terminasse antes de seu início seria vazia. É uma boa convenção, e se encaixava bem no modo como a classe funcionava. Logo depois de termos começado a usar essa convenção, percebemos que apenas criar um intervalo de datas que começasse após o seu término não era um código claro, portanto extraímos esse comportamento e colocamos em um método de factory para intervalos de data vazios.

> Havíamos feito essa alteração para deixar o código mais claro, mas tivemos uma recompensa inesperada. Criamos um intervalo de data vazio constante e adaptamos o método de factory para que devolvesse esse objeto em vez de criá-lo a cada vez. Essa mudança dobrou a velocidade do sistema, o suficiente para que os testes fossem suportáveis. Demoramos aproximadamente cinco minutos.
>
> Eu havia feito especulações com vários membros da equipe (Kent e Martin negam ter participado das especulações) sobre o que poderia estar provavelmente errado com o código que conhecíamos tão bem. Havíamos feito até mesmo um esboço de alguns designs para melhorias, sem antes avaliar o que estava acontecendo.
>
> Estávamos totalmente errados. Além de uma conversa realmente interessante, não estávamos fazendo nada benéfico.
>
> Eis a lição: mesmo que você saiba exatamente o que está acontecendo em seu sistema, faça medições de desempenho, não especule. Você aprenderá algo, e nove em cada dez vezes, não será o caso de você estar certo!
>
> — Ron Jeffries

O aspecto interessante sobre o desempenho é que, na maioria dos programas, a maior parte do tempo é gasta em uma pequena fração do código. Se eu otimizar todo o código igualmente, acabarei com 90% de meu trabalho desperdiçado porque estaria otimizando um código que não é muito executado. O tempo gasto para deixar o programa mais rápido – o tempo perdido por causa da falta de clareza – será todo desperdiçado.

A terceira abordagem para melhoria do desempenho tira proveito dessa estatística de 90%. Nessa abordagem, construo meu programa de modo bem fatorado, sem prestar atenção no desempenho, até iniciar um exercício deliberado de otimização visando ao desempenho. Durante essa otimização para o desempenho, sigo um processo específico para ajustar o programa.

Começo executando o programa em um profiler que o monitore e me informe em que pontos o programa está consumindo tempo e espaço. Desse modo, posso encontrar aquela pequena parte do programa em que estão os pontos determinantes para o desempenho. Então, foco nesses pontos determinantes usando as mesmas otimizações que utilizaria na abordagem da atenção constante. No entanto, como estou mantendo o foco de minha atenção em um ponto determinante, terei um efeito muito maior com menos

trabalho. Mesmo assim, continuo cauteloso. Assim como na refatoração, faço as alterações em passos pequenos. Depois de cada passo, compilo, testo e executo novamente o profiler. Caso o desempenho não tenha melhorado, desfaço a mudança. Continuo o processo de localizar e remover os pontos determinantes até obter um desempenho que satisfaça aos meus usuários.

Ter um programa bem fatorado ajuda nesse estilo de otimização de duas maneiras. Em primeiro lugar, me proporciona tempo para investir em ajustes que visem ao desempenho. Com um código bem fatorado, posso acrescentar funcionalidades com mais rapidez. Isso me dá mais tempo para focar no desempenho. (Executar um profiler garante que eu gaste esse tempo no lugar certo.) Em segundo lugar, com um programa bem fatorado, tenho mais especificidade em minha análise de desempenho. Meu profiler me conduz a partes menores do código, que são mais fáceis de ajustar. Com um código mais claro, tenho uma melhor compreensão acerca de minhas opções e dos tipos de ajustes que funcionarão.

Tenho percebido que a refatoração me ajuda a escrever softwares mais rápidos. Ela deixa o software mais lento no curto prazo, enquanto estou refatorando, porém facilita fazer ajustes durante a otimização. Acabo tendo uma boa vantagem no final.

De onde veio a refatoração?

Não fui bem-sucedido em determinar exatamente o nascimento do termo “refatoração”. Bons programadores sempre gastaram pelo menos um pouco de tempo limpando seu código. Eles fazem isso porque aprenderam que um código limpo é mais fácil de alterar do que um código complexo e confuso, e bons programadores sabem que raramente escrevem um código limpo na primeira vez.

A refatoração vai além. Neste livro, defendo a refatoração como um elemento essencial no processo de desenvolvimento de software como um todo. Duas das primeiras pessoas a reconhecer a importância da refatoração foram Ward Cunningham e Kent Beck, que trabalharam com Smalltalk a partir dos anos 1980. Smalltalk é um ambiente que, mesmo na época, era particularmente receptivo à refatoração. É um ambiente bem dinâmico, que permite escrever rapidamente um software altamente funcional. Smalltalk tinha um ciclo bem curto para compilação-linkagem-execução para a sua época, o que facilitava fazer alterações rapidamente, quando ciclos de compilações noturnas não eram desconhecidos. Smalltalk também é

orientado a objetos e, desse modo, oferece ferramentas eficazes para minimizar o impacto das alterações por trás de interfaces bem definidas. Ward e Kent exploraram abordagens de desenvolvimento de software voltadas para esse tipo de ambiente, e seu trabalho se desenvolveu dando origem à Extreme Programming. Eles perceberam que a refatoração era importante para melhorar a produtividade, e, desde então, vêm trabalhando com ela, de modo a aplicá-la em projetos de software sérios e aprimorá-la.

As ideias de Ward e Kent exerceram profunda influência na comunidade Smalltalk, e a noção de refatoração tornou-se um elemento importante na cultura de Smalltalk. Outra figura de liderança na comunidade Smalltalk é Ralph Johnson, um professor da Universidade de Illinois em Urbana-Champaign, famoso por ser um dos autores do livro da "Gangue dos Quatro" (Gang of Four) [gof] sobre padrões de projeto (design patterns). Um dos principais interesses de Ralph está no desenvolvimento de frameworks de software. Ele explorou o modo como a refatoração pode ajudar no desenvolvimento de um framework eficiente e flexível.

Bill Opdyke foi um dos alunos de doutorado de Ralph e estava particularmente interessado em frameworks. Ele viu o valor em potencial da refatoração e percebeu que ela poderia ser aplicada em muitos outros ambientes além de Smalltalk. Sua experiência anterior havia sido no desenvolvimento de centrais telefônicas, no qual uma boa dose de complexidade surge com o tempo e as mudanças são difíceis de fazer. A pesquisa de doutorado de Bill olhava a refatoração do ponto de vista de quem constrói uma ferramenta. Bill estava interessado em refatorações que fossem úteis no desenvolvimento de um framework C++; ele pesquisou as refatorações que preservavam a semântica necessária e mostrou como provar que elas a preservavam e como uma ferramenta poderia implementar essas ideias. A tese de doutorado de Bill [Opdyke] foi o primeiro trabalho significativo sobre refatoração.

Lembro-me de ter conhecido Bill na conferência OOPSLA em 1992. Sentamos juntos em um café e ele me contou sobre sua pesquisa. Lembro-me de ter pensado: "Interessante, mas não é realmente tão importante". Cara, como eu estava errado!

John Brant e Don Roberts levaram as ideias sobre ferramenta para refatoração muito além e criaram o Refactoring Browser, a primeira ferramenta de refatoração, adequada para o ambiente Smalltalk.

E eu? Sempre fui a favor de um código limpo, mas jamais havia

considerado isso tão importante. Então trabalhei em um projeto com Kent e vi o modo como ele usava a refatoração. Percebi a diferença que fazia em termos de produtividade e de qualidade. Essa experiência me convenceu de que a refatoração era uma técnica muito importante. Contudo, eu me sentia frustrado, pois não havia livro algum que eu pudesse dar a um programador que estivesse trabalhando, e nenhum dos experts mencionados anteriormente tinha qualquer plano para escrever um livro como esse. Assim, com a ajuda deles, eu fiz isso – o que levou à primeira edição deste livro.

Felizmente o conceito de refatoração se tornou comum no mercado. O livro teve boas vendas, e a refatoração entrou no vocabulário da maioria dos programadores. Outras ferramentas surgiram, especialmente para Java. Uma desvantagem dessa popularidade foi o fato de as pessoas usarem "refatoração" de forma genérica para indicar qualquer tipo de reestruturação. Apesar disso, ela se tornou uma prática convencional.

Refatorações automatizadas

Talvez a maior mudança na refatoração na última década, ou aproximadamente, esteja na disponibilidade de ferramentas com suporte para refatoração automatizada. Se eu quiser renomear um método em Java e estiver usando o IntelliJ IDEA [intellij] ou o Eclipse [eclipse] (para mencionar somente dois), posso fazer isso selecionando um item no menu. A ferramenta completa a refatoração para mim – e, em geral, tenho confiança suficiente em seu trabalho, a ponto de não me importar em executar a suíte de testes.

A primeira ferramenta que fez isso foi o Smalltalk Refactoring Browser, escrita por John Brandt e Don Roberts. A ideia foi rapidamente aceita na comunidade Java no início do século. Quando a JetBrains lançou seu IntelliJ IDEA IDE, a refatoração automatizada era um de seus recursos atraentes. A IBM a seguiu de perto logo depois, com ferramentas de refatoração como o Visual Age para Java. O Visual Age não teve grande impacto, mas muitos de seus recursos foram reimplementados no Eclipse, incluindo o suporte à refatoração.

A refatoração também chegou ao C#, inicialmente por meio do Resharper da JetBrains, um plug-in para o Visual Studio. Mais tarde, a equipe do Visual Studio acrescentou algumas funcionalidades para refatoração.

Atualmente, é muito comum encontrar algum tipo de suporte para refatoração nos editores e nas ferramentas, embora os recursos propriamente

ditos possam variar um pouco. Parte dessas variações se deve à ferramenta, enquanto outras são causadas por limitações quanto ao que é possível fazer com a refatoração automatizada em linguagens diferentes. Não analisarei as funcionalidades de diferentes ferramentas neste livro, mas acho que vale a pena falar um pouco sobre alguns dos princípios subjacentes.

Um modo genérico de automatizar uma refatoração é fazer manipulações de texto, por exemplo, busca/substituição para alterar um nome, ou alguma reorganização de código simples para *Extrair variável (Extract Variable)*. Essa é uma abordagem bem grosseira, na qual certamente não é possível confiar sem executar os testes novamente. Contudo, ela pode ser um primeiro passo conveniente. Uso macros desse tipo no Emacs para agilizar meu trabalho de refatoração quando não há refatorações mais sofisticadas à minha disposição.

Para fazer a refatoração de modo apropriado, a ferramenta deve trabalhar na árvore sintática do código, e não no texto. Manipular a árvore sintática é muito mais confiável para preservar o que o código faz. É por isso que, atualmente, a maioria dos recursos para refatoração faz parte de IDEs potentes – eles usam a árvore sintática não só para a refatoração, mas também para navegação no código, linting e atividades desse tipo. Essa colaboração entre texto e árvore sintática é o que faz com que os IDEs sejam mais que editores de texto.

Refatorar não é somente entender e atualizar a árvore sintática. A ferramenta também deve descobrir como renderizar o código novamente para texto na janela do editor. Considerando tudo isso, implementar uma refatoração razoável é um exercício de programação desafiador – um exercício do qual não estou ciente na maioria das vezes, quando uso despreocupadamente as ferramentas.

Muitas refatorações se tornam bem mais seguras quando aplicadas em uma linguagem com tipagem estática. Considere um simples *Renomear função (Rename Function)*. Posso ter métodos `addClient` em minhas classes `Salesman` e `Server`. Gostaria de renomear o método que está em meu salesman (vendedor), cujo propósito é diferente daquele que está em meu server (servidor), o qual não quero renomear. Sem uma tipagem estática, a ferramenta terá dificuldade em dizer se uma chamada a `addClient` foi feita para salesman. No Refactoring Browser, uma lista de pontos de chamada seria gerada, e eu decidiria manualmente quais eu gostaria de alterar. Isso faz com que essa refatoração não seja segura, forçando-me a executar os testes novamente. Uma

ferramenta como essa continua útil – mas a operação equivalente em Java pode ser totalmente segura e automática. Como a ferramenta é capaz de resolver o método para a classe correta com tipagem estática, posso ter a confiança de que a ferramenta alterará somente os métodos que devem ser alterados.

As ferramentas, em geral, vão além. Se eu renomear uma variável, posso ser questionado se mudanças em comentários que usam esse nome devem ser feitas. Se eu usar *Extrair função (Extract Function)*, a ferramenta identificará um código que duplique o corpo da nova função e se oferecerá para substituí-la por uma chamada. Programar com refatorações eficazes como essas é um motivo atraente para usar um IDE, em vez de se ater a um editor de texto familiar. Pessoalmente, sou um grande usuário do Emacs, mas, quando trabalho com Java, prefiro o IntelliJ IDEA ou o Eclipse – em grande parte, devido ao suporte à refatoração.

Embora ferramentas de refatoração sofisticadas sejam quase mágicas quanto à capacidade de refatorar um código de modo seguro, há alguns casos peculiares em que elas podem deixar a desejar. Ferramentas menos maduras têm dificuldade com chamadas reflexivas, como `Method.invoke` em Java (embora ferramentas mais maduras lidem muito bem com elas). Assim, mesmo com refatorações, em sua maior parte seguras, executar a suíte de testes com frequência para garantir que nada apresente problemas é uma atitude inteligente. Em geral, refatoro com uma combinação de refatorações automatizadas e manuais, portanto executo meus testes com frequência suficiente.

A capacidade de usar a árvore sintática para analisar e refatorar programas é uma vantagem atraente dos IDEs em comparação com os editores de texto simples, mas muitos programadores preferem a flexibilidade de seu editor de texto favorito, e gostaria de ter ambos. Uma tecnologia que atualmente vem ganhando impulso é a Language Servers [langserver]: um software que monta uma árvore sintática e apresenta uma API nos editores de texto. Esses servidores de linguagens podem aceitar vários editores de texto e disponibilizar comandos para uma análise de código e operações de refatoração sofisticadas.

Indo além

Pode parecer um pouco estranho falar sobre outras leituras já no segundo capítulo, mas este é um local tão bom quanto qualquer outro para enfatizar

que há outros materiais por aí sobre refatoração que vão além do básico que está nesta obra.

Este livro ensinou refatoração a muitas pessoas, mas meu foco está mais voltado em apresentar uma referência para a refatoração do que em conduzir os leitores no processo de aprendizado. Se você estiver procurando um livro como esse, sugiro a obra *Refactoring Workbook* [Wake] de Bill Wake, que contém muitos exercícios para praticar a refatoração.

Muitos dos que foram pioneiros na refatoração também eram ativos na comunidade de padrões de software. Josh Kerievsky aproximou bastante esses dois mundos com o livro *Refatoração para padrões* [Kerievsky], que descreve os padrões mais importantes do extremamente influente livro da "Gangue dos Quatro" [gof] e mostra como usar a refatoração para evoluir em direção a eles.

Este livro tem como foco a refatoração na programação de propósito geral, mas ela também pode ser aplicada em áreas especializadas. Duas obras que receberam bastante atenção são *Refactoring Databases* [Ambler & Sadalage] (de Scott Ambler e Pramod Sadalage) e *Refatorando HTML* [Harold] (de Elliotte Rusty Harold).

Embora não tenha refatoração no título, também vale a pena incluir *Trabalho eficaz com código legado* de Michael Feathers [Feathers], que é um livro essencialmente sobre como pensar em refatorar uma base de código antiga com uma cobertura de testes precária.

Apesar de este livro (e seu antecessor) estar voltado para programadores de qualquer linguagem, há lugar para livros de refatoração em linguagens específicas. Dois de meus antigos colegas, Jay Fields e Shane Harvey, fizeram isso para a linguagem de programação Ruby [Fields et al.].

Para um conteúdo mais atualizado, consulte o representante deste livro na web, bem como o site principal sobre refatoração: refactoring.com [ref.com].

---

1 N.T.: Em inglês, tanto o substantivo como o verbo correspondem a um único termo, *refactoring*.

CAPÍTULO 3

“Maus cheiros” no código

por Kent Beck e Martin Fowler

Se está cheirando mal, troque-o.
– Vovó Beck, discutindo a filosofia da criação de filhos

A essa altura, você tem uma boa ideia de como a refatoração funciona. Mas apenas saber como ela funciona não significa que você saiba quando deve fazê-la. Decidir quando começar a refatorar – e quando parar – é tão importante para a refatoração quanto saber como lidar com o seu modo de funcionamento.

Estamos agora diante de um dilema. É fácil explicar como remover uma variável de instância ou criar uma hierarquia. São questões simples. Tentar explicar *quando* você deve fazer isso não é tão simples assim. Em vez de apelar para alguma vaga noção de estética de programação (que, francamente, é o que nós consultores em geral fazemos), gostaria de ter algo um pouco mais sólido.

Quando eu estava escrevendo a primeira edição deste livro, fiquei remoendo esse problema enquanto visitava Kent Beck em Zurique. Talvez ele estivesse sob a influência dos odores de sua filha recém-nascida na época, mas havia concebido a noção de descrever o “quando” da refatoração em termos de cheiros.

“Cheiros”, você diria, “e isso deveria ser melhor do que uma estética vaga?” Bem, sim. Já vimos muitos códigos, escritos para projetos que variam por toda gama deles, desde os extremamente bem-sucedidos até os quase mortos. Ao fazer isso, aprendemos a procurar determinadas estruturas no código que sugerem – às vezes, clamam – a possibilidade de refatoração. (Estamos passando para “nós” neste capítulo para refletir o fato de que Kent e eu o escrevemos juntos. Você pode notar a diferença porque as piadas engraçadas são minhas e as demais são dele.)

Uma informação que não tentaremos dar a você são os critérios exatos para

decidir quando uma refatoração já deveria ter sido feita. De acordo com nossa experiência, nenhum conjunto de métricas é páreo para a intuição humana bem fundamentada. O que faremos é apresentar indícios a você de que há um problema que pode ser resolvido com uma refatoração. Você terá de desenvolver o próprio senso para saber quantas variáveis de instância ou quantas linhas de código em um método são demais.

Use este capítulo como uma forma de se sentir inspirado quando não tiver certeza das refatorações que deve fazer. Leia o capítulo (ou passe os olhos pela tabela) e tente identificar o cheiro que está sentindo, e então consulte as refatorações que sugerimos para ver se elas poderão ajudar. Talvez você não encontre o cheiro exato a ser detectado, mas espero que possamos colocar você na direção correta.

Nome misterioso

Quebrar a cabeça diante de algum texto para entender o que está acontecendo é ótimo se você estiver lendo um romance policial, mas não se estiver lendo código. Podemos fantasiar pensando que somos um agente secreto especial, mas nosso código deve ser mundano e claro. Uma das partes mais importantes de um código claro são nomes bons, portanto devemos pensar bem ao nomear funções, módulos, variáveis e classes, de modo que eles comuniquem claramente o que fazem e como são usados.

Infelizmente, porém, nomear é uma das duas tarefas mais difíceis [mf-2h] em programação. Assim, talvez as refatorações mais comuns que façamos sejam para renomear algo: *Mudar declaração de função (Change Function Declaration)* (para renomear uma função), *Renomear variável (Rename Variable)* e *Renomear campo (Rename Field)*. Muitas vezes, as pessoas têm medo de renomear itens, achando que não vale a pena se dar ao trabalho, mas um bom nome pode fazer você economizar horas que seriam gastas quebrando a cabeça por falta de compreensão, no futuro.

Renomear não é somente um exercício de mudança de nomes. Quando você não é capaz de pensar em um bom nome para algo, em geral é sinal de que há um problema mais profundo no design. Quebrar a cabeça diante de um nome complicado muitas vezes nos levou a simplificações significativas em nosso código.

Código duplicado

Se você vir a mesma estrutura de código em mais de um lugar, pode ter certeza de que seu programa será melhor se encontrar uma forma de unificar essas estruturas. Uma duplicação significa que, sempre que ler essas cópias, você terá de fazer isso cuidadosamente para ver se há alguma diferença. Se houver necessidade de alterar o código duplicado, você deverá localizar e considerar cada duplicação.

O problema mais simples de código duplicado ocorre quando a mesma expressão está em dois métodos da mesma classe. Então, tudo que você tem a fazer é usar *Extrair função (Extract Function)* e chamar o código nos dois lugares. Se você tiver um código que seja parecido, mas não exatamente idêntico, veja se é possível usar *Deslocar instruções (Slide Statements)* para organizar o código de modo que os itens semelhantes estejam todos juntos para facilitar a extração. Se os fragmentos duplicados estiverem em subclasses de uma classe-base comum, você poderá usar *Subir método (Pull Up Method)* para evitar chamar um a partir do outro.

Função longa

De acordo com a nossa experiência, os programas que vivem melhor e por mais tempo são aqueles que têm funções pequenas. Programadores para quem uma base de código como essa é uma novidade muitas vezes acham que nenhum processamento ocorre – acham que o programa é uma sequência interminável de delegações. Contudo, depois de conviver com um programa desse tipo por alguns anos, você perceberá quão importantes são essas funções pequenas. Todas as recompensas pelos acessos indiretos – explicação, compartilhamento e escolha – recebem suporte das funções pequenas.

Desde o início da era da programação, as pessoas perceberam que, quanto mais longa uma função, mais difícil será compreendê-la. Linguagens mais antigas tinham um overhead para as chamadas de sub-rotinas, o que desencorajava as pessoas a usar funções pequenas. As linguagens modernas praticamente eliminaram esse overhead para chamadas in-process. Continua havendo um overhead para o leitor do código porque é necessário mudar de contexto para ver o que a função faz. Ambientes de desenvolvimento que permitem alternar rapidamente de uma chamada de função para a sua declaração, ou permitam ver as duas funções ao mesmo tempo, ajudam a eliminar esse passo; no entanto, o verdadeiro segredo para facilitar a

compreensão de funções pequenas é uma boa nomenclatura. Se você atribuir um bom nome para uma função, basicamente não será necessário olhar para o corpo dela.

O efeito final é que você deve ser muito mais agressivo no que concerne à decomposição de funções. Um método heurístico que seguimos é que, sempre que sentimos a necessidade de comentar algo, escrevemos uma função em seu lugar. Uma função como essa contém o código que queríamos comentar, mas recebe o nome com base na *intenção* do código, em vez do modo como ele funciona. Podemos fazer isso com um grupo de linhas ou até mesmo com uma única linha de código. Fazemos isso mesmo que a chamada do método seja mais longa que o código substituído – desde que o nome do método explique o propósito do código. O segredo, nesse caso, não é o tamanho da função, mas a distância semântica entre o que o método faz e como ele faz.

Em 99% do tempo, tudo que você tem a fazer para reduzir uma função é usar *Extrair função (Extract Function)*. Encontre partes da função que pareçam ficar bem se estiverem juntas e crie uma nova função.

Se tiver uma função com muitos parâmetros e variáveis temporárias, eles atrapalharão a extração. Se tentar usar *Extrair função (Extract Function)*, você acabará passando parâmetros demais para o método extraído, a ponto de o resultado dificilmente ser mais legível que o original. Com frequência, poderá usar *Substituir variável temporária por consulta (Replace Temp with Query)* para eliminar as variáveis temporárias. Listas longas de parâmetros podem ser reduzidas com *Introduzir objeto de parâmetros (Introduce Parameter Object)* e *Preservar objeto inteiro (Preserve Whole Object)*.

Se você tentou isso e ainda continua com muitas variáveis temporárias e parâmetros, é hora de trazer a artilharia pesada: *Substituir função por comando (Replace Function with Command)*.

Como podemos identificar as porções de código a serem extraídas? Uma boa técnica é procurar comentários. Com frequência, eles sinalizam esse tipo de distância semântica. Um bloco de código com um comentário que informe o que ele faz pode ser substituído por um método cujo nome seja baseado no comentário. Vale a pena extrair até mesmo uma única linha caso ela precise de explicações.

As condicionais e os laços também dão sinais de extrações. Use *Decompor condicional (Decompose Conditional)* para lidar com expressões

condicionais. Uma instrução switch longa deve ter seus ramos transformados em chamadas de funções únicas com *Extrair função (Extract Function)*. Se houver mais de uma instrução switch para a mesma condição, você deverá aplicar *Substituir condicional por polimorfismo (Replace Conditional with Polymorphism)*.

Com laços, extraia o laço e o código dentro do laço em seu próprio método. Se achar difícil dar um nome a um laço extraído, talvez seja pelo fato de ele estar fazendo duas tarefas distintas – nesse caso, não tenha medo de usar *Dividir laço (Split Loop)* para separar essas tarefas.

Lista longa de parâmetros

No início da era da programação, éramos ensinados a passar tudo que fosse necessário a uma função como parâmetros. Isso era compreensível, pois a alternativa seria usar dados globais, e dados globais rapidamente se tornam uma ameaça. Porém, listas longas de parâmetros muitas vezes são confusas por si só.

Se puder obter um parâmetro solicitando-o a outro parâmetro, *Substituir parâmetro por consulta (Replace Parameter with Query)* poderá ser usado para eliminar o segundo parâmetro. Em vez de extrair vários dados de uma estrutura de dados existente, você pode usar *Preservar objeto inteiro (Preserve Whole Object)* e passar a estrutura de dados original. Se vários parâmetros sempre estiverem juntos, combine-os usando *Introduzir objeto de parâmetros (Introduce Parameter Object)*. Se um parâmetro for usado como uma flag para acionar diferentes comportamentos, use *Remover argumento de flag (Remove Flag Argument)*.

Classes são uma ótima maneira de reduzir tamanhos de listas de parâmetros. Elas são particularmente convenientes quando várias funções compartilham diversos valores de parâmetros. Então você pode usar *Combinar funções em classe (Combine Functions into Class)* para capturar esses valores comuns como campos. Se colocarmos nossos chapéus de programação funcional, diríamos que isso cria um conjunto de funções parcialmente aplicadas.

Dados globais

Desde que começamos a escrever software, somos avisados dos perigos dos dados globais – como foram inventados por demônios da quarta dimensão do

inferno, que é o local de descanso de qualquer programador que ouse utilizá-los. E, embora sejamos um tanto quanto céticos quanto a fogo e enxofre, esse ainda é um dos odores mais pungentes com o qual provavelmente deparamos. O problema com um dado global é que ele pode ser modificado em qualquer ponto da base de código, e não há nenhum esquema para descobrir qual parte do código fez a alteração. Ocasionalmente, isso resulta em bugs que surgem de uma espécie de ação sinistra à distância – e é muito difícil descobrir onde está a porção culpada do programa. A forma mais evidente de dados globais são as variáveis globais, mas também vemos esse problema em variáveis de classe e em singletons.

Nossa principal defesa nesse caso é *Encapsular variável (Encapsulate Variable)*, que é sempre o primeiro passo quando somos confrontados com dados abertos à contaminação por qualquer parte de um programa. Pelo menos, quando você os tiver encapsulados por uma função, poderá começar a ver em que lugar esses dados são modificados e começar a controlar o seu acesso. Portanto, é bom limitar seu escopo o máximo possível movendo-os para uma classe ou módulo, de modo que somente o código desse módulo possa vê-los.

Dados globais são particularmente inconvenientes quando mutáveis. Dados globais para os quais seja possível garantir que não serão alterados depois que o programa começar são relativamente seguros – se você tiver uma linguagem capaz de impor essa garantia.

Os dados globais ilustram a máxima de Paracelsus: a diferença entre um veneno e algo benigno está na dose. Você pode não ter problemas com pequenas doses de dados globais, mas será exponencialmente mais difícil lidar com eles, quanto mais dados desse tipo você tiver. Mesmo com poucos dados globais, gostamos de mantê-los encapsulados – esse é o segredo para lidar com mudanças à medida que o software evoluir.

Dados mutáveis

Mudanças em dados muitas vezes podem resultar em consequências inesperadas e bugs complicados. Posso atualizar alguns dados aqui sem perceber que outra parte do software espera algo diferente e agora falha – uma falha particularmente difícil de localizar caso ela ocorra apenas em condições raras. Por esse motivo, toda uma escola de desenvolvimento de software – a programação funcional – está baseada na noção de que os dados

jamais devem mudar e que a atualização de uma estrutura de dados sempre deve devolver uma nova cópia da estrutura com a mudança, deixando os dados antigos intocados.

Esses tipos de linguagens, porém, ainda são uma parte relativamente pequena na programação; muitos de nós trabalhamos com linguagens que permitem que as variáveis mudem. Entretanto, isso não significa que devemos ignorar as vantagens da imutabilidade – ainda há várias atitudes que podemos tomar para limitar os riscos em atualizações de dados irrestritas.

Você pode usar *Encapsular variável (Encapsulate Variable)* para garantir que todas as atualizações ocorram por meio de funções limitantes, mais simples de monitorar e que evoluem mais facilmente. Se uma variável for atualizada para armazenar dados diferentes, use *Separar variável (Split Variable)* tanto para mantê-las separadas quanto para evitar a atualização arriscada. Tente o máximo possível mover a lógica para fora do código que processa a atualização usando *Deslocar instruções (Slide Statements)* e *Extrair função (Extract Function)* a fim de separar o código livre de efeitos colaterais de qualquer parte que faça a atualização. Em APIs, use *Separar consulta de modificador (Separate Query from Modifier)* para garantir que quem faz a chamada não precise chamar o código que tenha efeitos colaterais, a menos que seja realmente necessário. Gostamos de usar *Remover método de escrita (Remove Setting Method)* assim que possível – às vezes, somente tentar encontrar os clientes de um setter ajuda a identificar oportunidades para reduzir o escopo de uma variável.

Dados mutáveis que podem ser calculados em outros lugares são particularmente pungentes. Não são apenas uma fonte profícua de confusão, bugs e jantares perdidos em casa – também são desnecessários. Passamos neles um spray com uma solução concentrada de vinagre e usamos *Substituir variável derivada por consulta (Replace Derived Variable with Query)*.

Um dado mutável não é um grande problema se for uma variável cujo escopo tenha apenas algumas linhas – mas seu risco aumentará à medida que o escopo se ampliar. Use *Combinar funções em classe (Combine Functions into Class)* ou *Combinar funções em transformação (Combine Functions into Transform)* para limitar a quantidade de código que precise atualizar uma variável. Se uma variável contiver um dado com estrutura interna, em geral será melhor substituir a estrutura toda em vez de modificá-la in-place usando *Mudar referência para valor (Change Reference to Value)*.

Alteração divergente

Estruturamos nosso software de modo a facilitar as alterações; afinal de contas, um software deve ser soft. Quando fazemos uma alteração, queremos ser capazes de acessar diretamente um único ponto claro no sistema e fazer essa alteração. Se não puder fazer isso, é sinal de que estará sentindo dois cheiros pungentes intimamente relacionados.

Uma alteração divergente ocorre quando um módulo é frequentemente alterado de formas diferentes por motivos diferentes. Se você olhar para um módulo e disser: "Bem, terei que modificar essas três funções sempre que tiver um novo banco de dados; tenho que modificar essas quatro funções sempre que houver um novo instrumento financeiro", essa é uma indicação de uma alteração divergente. A interação com o banco de dados e os problemas de processamento financeiro são contextos diferentes, e podemos melhorar nossa vida de programadores se passarmos esses contextos para módulos separados. Desse modo, quando houver uma alteração em um contexto, teremos de entender somente esse contexto único e poderemos ignorar o outro. Sempre achamos isso importante; agora, porém, com nossos cérebros encolhendo com a idade, isso se torna mais imperativo ainda. É claro que, com frequência, você perceberá a situação apenas depois de ter adicionado alguns bancos de dados ou instrumentos financeiros; as fronteiras dos contextos em geral não são claras no início de um programa e continuarão a se deslocar à medida que as funcionalidades de um sistema de software mudarem.

Se os dois aspectos formarem naturalmente uma sequência – por exemplo, você obtém dados do banco de dados e então aplica seu processamento financeiro neles –, então *Separar em fases (Split Phase)* separará ambos com uma estrutura de dados clara entre eles. Se houver mais idas e vindas nas chamadas, crie módulos apropriados e use *Mover função (Move Function)* para separar o processamento. Se as funções misturarem os dois tipos de processamento internamente, use *Extrair função (Extract Function)* para separá-las antes de movê-las. Se os módulos forem classes, então *Extrair classe (Extract Class)* ajudará a formalizar o modo de fazer a separação.

Cirurgia com rifle

Uma cirurgia com rifle é parecida com uma alteração divergente, porém é o oposto. Você sentirá o seu odor quando, sempre que fizer uma alteração, tiver

de fazer várias modificações pequenas em várias classes diferentes. Quando as alterações estão por toda parte, elas são difíceis de encontrar, e será fácil se esquecer de uma modificação importante.

Nesse caso, você deve usar *Mover função (Move Function)* e *Mover campo (Move Field)* para colocar todas as alterações em um único módulo. Se tiver um conjunto de funções atuando em dados similares, use *Combinar funções em classe (Combine Functions into Class)*. Se tiver funções que transformem ou enriqueçam uma estrutura de dados, use *Combinar funções em transformação (Combine Functions into Transform)*. *Separar em fases (Split Phase)* muitas vezes será conveniente nesse caso se as funções comuns puderem combinar suas saídas, fornecendo-as a uma fase de consumo da lógica.

Uma tática útil em uma cirurgia com rifle é usar refatorações de internalização, como *Internalizar função (Inline Function)* ou *Internalizar classe (Inline Class)*, a fim de reunir uma lógica indevidamente separada. Você acabará com um Método Longo ou uma Classe Grande, mas poderá então usar extrações para separá-los em partes mais razoáveis. Apesar de sermos extremamente afeiçoados a funções e classes pequenas em nosso código, não temos medo de criar algo grande como um passo intermediário na reorganização.

Inveja de recursos

Quando modularizamos um programa, estamos tentando separar o código em zonas para maximizar a interação dentro de uma zona e minimizá-la entre zonas diferentes. Um caso clássico de Inveja de Recursos ocorre quando uma função em um módulo gasta mais tempo se comunicando com funções ou dados de outro módulo do que com o próprio módulo. Perdemos conta das vezes que vimos uma função chamar meia dúzia de métodos getter em outro objeto para calcular algum valor. Felizmente a cura para esse caso é óbvia: a função claramente quer estar com os dados, portanto use *Mover função (Move Function)* para levá-la para lá. Às vezes, apenas parte de uma função sofre de inveja, e, nesse caso, você deve usar *Extrair função (Extract Function)* na parte invejosa, e *Mover função (Move Function)* para lhe dar um lar dos sonhos.

É claro que nem todos os casos são simples e diretos. Com frequência, uma

função usa recursos de vários módulos, portanto em qual deles ela deve morar? O método heurístico que usamos consiste em determinar qual módulo tem a maior parte dos dados e colocamos a função com esses dados. Esse passo muitas vezes será mais fácil se você usar *Extrair função (Extract Function)* para separar a função em partes que serão colocadas em lugares diferentes.

É claro que há vários padrões sofisticados que violam essa regra. Da Gangue dos Quatro [gof], Strategy (Estratégia) e Visitor (Visitante) logo vêm à mente. Self Delegation (Autodelegação) de Kent Beck [Beck SBPP] é outro. Use-os para combater o cheiro da alteração divergente. A regra geral básica é colocar junto aquilo que muda junto. Dados e os comportamentos que referenciam esses dados em geral mudam juntos – mas há exceções. Quando as exceções ocorrem, movemos o comportamento para manter as alterações em um só lugar. Strategy e Visitor lhe permitem modificar facilmente o comportamento porque isolam a pequena dose de comportamento que deve ser sobrescrita, ao custo de mais acessos indiretos.

Agrupamentos de dados

Dados tendem a ser como crianças: gostam de ficar juntos. Com frequência, você verá os mesmos três ou quatro dados juntos em vários lugares: como campos em algumas classes, como parâmetros em várias assinaturas de métodos. Conjuntos de dados que andam juntos realmente deveriam ter um lar juntos. O primeiro passo é procurar os lugares em que esses amontoados de dados aparecem como campos. Use *Extrair classe (Extract Class)* nos campos para transformar os agrupamentos em um objeto. Em seguida, volte sua atenção para as assinaturas de métodos usando *Introduzir objeto de parâmetros (Introduce Parameter Object)* ou *Preservar objeto inteiro (Preserve Whole Object)* para reduzi-los. A vantagem imediata é a possibilidade de reduzir muitas listas de parâmetros e simplificar as chamadas de métodos. Não se preocupe com os agrupamentos de dados que usem somente alguns dos campos do novo objeto. Desde que você esteja substituindo dois ou mais campos pelo novo objeto, ainda será vantajoso.

Um bom teste é considerar o que aconteceria se você apagasse um dos dados. Se fizer isso, os demais fariam sentido? Se não fizerem, é um sinal evidente de que você tem um objeto que está implorando para nascer.

Você perceberá que defendemos a criação de uma classe nesse caso, e não

de uma simples estrutura de registro. Fazemos isso porque usar uma classe lhe dá a oportunidade de criar um perfume agradável. Agora você pode procurar casos de inveja de recursos, que sugerirão comportamentos possíveis de serem movidos para suas novas classes. Frequentemente, vemos isso como uma dinâmica poderosa, capaz de criar classes úteis e remover muita duplicação, além de agilizar desenvolvimentos futuros, permitindo que os dados se tornem membros produtivos da sociedade.

Obsessão por primitivos

A maioria dos ambientes de programação é desenvolvida com base em um conjunto amplamente usado de tipos primitivos: inteiros, números de ponto flutuante e strings. As bibliotecas podem acrescentar outros objetos pequenos, como datas. Percebemos que muitos programadores são curiosamente relutantes em criar os próprios tipos fundamentais que sejam úteis para o seu domínio – por exemplo moeda, coordenadas ou intervalos. Desse modo, vemos cálculos que tratam valores monetários como números simples, ou cálculos de quantidades físicas que ignoram as unidades (somando polegadas com milímetros) ou muito código executando `if (a < upper && a > lower)`.

Strings são particularmente placas de Petri comuns para esse tipo de odor: um número de telefone é mais que apenas um conjunto de caracteres. No mínimo, um tipo apropriado em geral é capaz de incluir uma lógica de exibição consistente quando tiver de ser apresentado em uma interface de usuário. Representar tipos como esses como strings é um cheiro desagradável tão comum que as pessoas os chamam de “tipados como string” (stringly typed).

Você pode sair da caverna primitiva e ir para o mundo com aquecimento central dos tipos significativos usando *Substituir primitivo por objeto (Replace Primitive with Object)*. Se o primitivo for um código de tipo que controle um comportamento condicional, use *Substituir código de tipos por subclasses (Replace Type Code with Subclasses)*, seguido de *Substituir condicional por polimorfismo (Replace Conditional with Polymorphism)*.

Grupos de primitivos que apareçam habitualmente juntos formam amontoados de dados e devem ser civilizados com *Extrair classe (Extract Class)* e *Introduzir objeto de parâmetros (Introduce Parameter Object)*.

Switches repetidos

Converse com um verdadeiro evangelista de orientação a objetos, e essas pessoas logo falarão dos males das instruções switch. Elas argumentarão que qualquer instrução switch que você vir estará implorando por *Substituir condicional por polimorfismo (Replace Conditional with Polymorphism)*. Já ouvimos até algumas pessoas argumentarem que toda lógica condicional deveria ser substituída por polimorfismo, jogando fora a maior parte dos ifs na lata de lixo da história.

Mesmo em nossa mais atordoada juventude, jamais fomos incondicionalmente contrários à condicional. Na verdade, a primeira edição deste livro continha um cheiro intitulado "instruções switch". O cheiro estava lá porque, no final dos anos 1990, percebemos que o polimorfismo estava tristemente desvalorizado, e vimos vantagem em fazer as pessoas efetuarem a troca.

Atualmente, o polimorfismo está mais presente, e não é mais tão raro quanto era quinze anos atrás. Além do mais, muitas linguagens oferecem suporte para formas mais sofisticadas de instruções switch, que usam mais que um código primitivo como base. Assim, nosso foco agora está no switch repetido, em que a mesma lógica condicional de switch (seja em uma instrução switch/case ou em uma cascata de instruções if/else) surge em diferentes lugares. O problema com switches duplicados como esse é que, sempre que você adicionar uma cláusula, será necessário encontrar todos os switches e atualizá-los. Contra as forças ocultas de tais repetições, o polimorfismo oferece uma arma elegante para uma base de código mais civilizada.

Laços

Os laços têm sido uma parte essencial da programação desde as primeiras linguagens. Contudo, achamos que eles são tão relevantes hoje em dia quanto as calças boca-de-sino e os papéis de parede com textura. Nós os tratamos com desdém na primeira edição – mas Java, assim como a maioria das demais linguagens na época, não oferecia nenhuma alternativa melhor. Atualmente, porém, funções de primeira classe são amplamente aceitas, portanto podemos usar *Substituir laço por pipeline (Replace Loop with Pipeline)* para aposentar esses anacronismos. Achamos que as operações de pipeline, como filtrar e mapear, nos ajudam a ver rapidamente os elementos

incluídos no processamento e o que é feito com eles.

Elemento ocioso

Gostamos de usar elementos de programa para agregar estrutura – oferecendo oportunidades para variação, reutilização ou somente para ter nomes mais convenientes. Às vezes, porém, a estrutura não é necessária. Podemos ter uma função cujos nome e código do corpo sejam iguais, ou uma classe que contém essencialmente uma única função simples. Em algumas ocasiões, esperávamos que a função fosse crescer e ser popular no futuro, mas ela acabou não realizando seus sonhos. Em outras, é uma classe cuja existência costumava valer a pena, porém foi rebaixada com a refatoração. Qualquer que seja o caso, esses elementos de programa devem morrer com dignidade. Em geral, isso significa usar *Internalizar função (Inline Function)* ou *Internalizar classe (Inline Class)*. Com herança, você pode usar *Condensar Hierarquia (Collapse Hierarchy)*.

Generalidade especulativa

Brian Foote sugeriu esse nome para um cheiro ao qual somos muito sensíveis. Você o sente quando as pessoas dizem: “Ah, eu acho que precisaremos desse tipo de coisa algum dia”, e então adicionam todo tipo de ganchos (hooks) e de casos especiais para lidar com casos que não são necessários. Com frequência, o resultado é mais difícil de entender e de manter. Se todos esses recursos estivessem sendo usados, valeria a pena. Mas, se não estiverem, não vale. Esses recursos simplesmente atrapalham, portanto livre-se deles.

Se você tiver classes abstratas que não façam muito, use *Condensar Hierarquia (Collapse Hierarchy)*. Uma delegação desnecessária pode ser removida com *Internalizar função (Inline Function)* e *Internalizar classe (Inline Class)*. Funções com parâmetros não usados devem ser submetidas a *Mudar declaração de função (Change Function Declaration)* a fim de remover esses parâmetros. Você também deve aplicar *Mudar declaração de função (Change Function Declaration)* para remover qualquer parâmetro desnecessário, que muitas vezes são criados para variações futuras que jamais chegam a existir.

Uma generalidade especulativa pode ser identificada quando os únicos

usuários de uma função ou de uma classe são os casos de teste. Se você encontrar algo desse tipo, apague o caso de teste e aplique *Remover código morto (Remove Dead Code)*.

Campo temporário

Às vezes, você vê uma classe em que um campo é definido somente em determinadas circunstâncias. Um código como esse é difícil de entender porque você espera que um objeto precise de todos os seus campos. Tentar entender por que um campo existe quando parece que não está sendo usado pode deixar você louco.

Use *Extrair classe (Extract Class)* para criar um lar para as pobres variáveis órfãs. Utilize *Mover função (Move Function)* para levar todo o código que diz respeito aos campos para essa nova classe. Você também pode eliminar código condicional usando *Introduzir caso especial (Introduce Special Case)* a fim de criar uma classe alternativa para quando as variáveis não forem válidas.

Cadeias de mensagens

Vemos cadeias de mensagens quando um cliente pede um objeto para outro objeto, cujo cliente então pede para outro objeto, cujo cliente então pede para outro objeto, e assim por diante. Você pode ver isso como uma longa linha de métodos getThis, ou como uma sequência de variáveis temporárias. Navegar dessa forma significa que o cliente está acoplado à estrutura da navegação. Qualquer alteração nos relacionamentos intermediários fará com que o cliente tenha de mudar.

O passo a ser dado nesse caso é *Ocultar delegação (Hide Delegate)*. Você pode fazer isso em vários pontos da cadeia. Em princípio, pode fazê-lo em qualquer objeto da cadeia, mas, em geral, cada objeto intermediário se transformará em um mediador (middle man). Muitas vezes, uma alternativa melhor é verificar para que o objeto resultante é usado. Veja se é possível usar *Extrair função (Extract Function)* para separar uma porção do código que use o objeto, e então *Mover função (Move Function)* para inseri-la na cadeia. Se vários clientes de um dos objetos da cadeia quiserem navegar pelo restante do caminho, acrescente um método para isso.

Algumas pessoas consideram qualquer cadeia de métodos como algo muito

ruim. Somos conhecidos por nossa calma e ponderada moderação. Bem, pelo menos nesse caso, nós somos.

Intermediário

Uma das principais características da orientação a objetos é o encapsulamento – ocultar detalhes internos do restante do mundo. O encapsulamento com frequência vem acompanhado da delegação. Você pergunta a uma diretora se ela está livre para uma reunião; ela delega a mensagem para a sua agenda e dá uma resposta a você. Até agora, tudo bem. Não há necessidade de saber se a diretora usa uma agenda, um dispositivo eletrônico ou uma secretária para administrar seus compromissos.

Contudo, isso pode ir longe demais. Você olha para a interface de uma classe e descobre que metade dos seus métodos está delegando para a outra classe. Depois de um tempo, é hora de usar *Remover intermediário (Remove Middle Man)* e conversar com o objeto que realmente sabe o que está acontecendo. Se houver poucos métodos que não estão fazendo muita coisa, utilize *Internalizar função (Inline Function)* para internalizá-los nos pontos em que as chamadas são feitas. Se houver comportamentos adicionais, você poderá usar *Substituir superclasse por delegação (Replace Superclass with Delegate)* ou *Substituir subclasse por delegação (Replace Subclass with Delegate)* para transformar o intermediário no objeto real. Isso permite estender o comportamento sem ter de perseguir toda essa delegação.

Trocas escusas

Pessoas que trabalham com software gostam de paredes robustas entre seus módulos e reclamam muito de como muitas trocas de dados aumentam o acoplamento. Para que um sistema funcione, algumas trocas têm de ocorrer, mas precisamos reduzi-las a um nível mínimo e mantê-las às claras.

Módulos que cochicham entre si na máquina de café devem ser separados usando *Mover função (Move Function)* e *Mover campo (Move Field)* para reduzir a necessidade de conversa. Se os módulos tiverem interesses comuns, experimente criar um terceiro módulo para manter essa parte comum em um local mais apropriado, ou use *Ocultar delegação (Hide Delegate)* para fazer outro módulo atuar como um intermediário.

A herança muitas vezes pode levar a um conluio. As subclasses sempre

saberão mais sobre seu pais do que seus pais gostariam que elas soubessem. Se é chegada a hora de sair de casa, aplique *Substituir subclasse por delegação (Replace Subclass with Delegate)* ou *Substituir superclasse por delegação (Replace Superclass with Delegate)*.

Classe grande

Quando uma classe tenta fazer demais, com frequência isso se evidencia como campos demais. Quando uma classe tem campos demais, um código duplicado não deve estar muito longe.

Você pode *Extrair classe (Extract Class)* para reunir algumas das variáveis. Escolha variáveis que façam sentido estarem reunidas em um componente. Por exemplo, "depositAmount" e "depositCurrency" provavelmente pertencem ao mesmo componente. De modo geral, prefixos ou sufixos comuns para um subconjunto das variáveis de uma classe sugerem a oportunidade para um componente. Se o componente fizer sentido com herança, você perceberá que *Extrair superclasse (Extract Superclass)* ou *Substituir código de tipos por subclasses (Replace Type Code with Subclasses)* (que, essencialmente, faz a extração de uma subclasse) em geral são mais fáceis de usar.

Às vezes, uma classe não utiliza todos os seus campos em todas as ocasiões. Nesse caso, você poderá fazer essas extrações várias vezes.

Como ocorre com uma classe com muitas variáveis de instância, uma classe com código demais é um terreno muito fértil para código duplicado, caos e morte. A solução mais simples (já mencionamos que gostamos de soluções simples?) é eliminar a redundância na própria classe. Se você tiver cinco métodos de cem linhas, com muito código em comum, talvez possa transformá-los em cinco métodos de dez linhas com outros dez métodos de duas linhas extraídos do original.

Os clientes de uma classe como essa muitas vezes são a melhor pista para dividir a classe. Observe se os clientes usam um subconjunto dos recursos da classe. Cada subconjunto é uma possível classe distinta. Depois de ter identificado um subconjunto conveniente, use *Extrair classe (Extract Class)*, *Extrair superclasse (Extract Superclass)* ou *Substituir código de tipos por subclasses (Replace Type Code with Subclasses)* para separá-la.

Classes alternativas com interfaces diferentes

Uma das grandes vantagens de usar classes é o suporte à substituição, permitindo que uma classe seja trocada por outra quando houver necessidade. Porém, isso só funcionará se suas interfaces forem as mesmas. Use *Mudar declaração de função (Change Function Declaration)* para fazer com que as funções coincidam. Com frequência, isso não será suficiente; continue usando *Mover função (Move Function)* para passar comportamentos para as classes até que os protocolos sejam iguais. Se isso resultar em duplicação, você poderá usar *Extrair superclasse (Extract Superclass)* para corrigir.

Classe de dados

Essas são as classes que têm campos, métodos de leitura e de escrita para os campos, e nada mais. Classes desse tipo são armazenadores de dados burros e, com frequência, são manipuladas de forma excessivamente detalhada por outras classes. Em alguns estágios, essas classes podem ter campos públicos. Nesse caso, você deve aplicar imediatamente *Encapsular registro (Encapsulate Record)* antes que alguém perceba. Use *Remover método de escrita (Remove Setting Method)* de qualquer campo que não deva ser alterado.

Observe em que locais esses métodos de leitura e de escrita estão sendo usados por outras classes. Tente usar *Mover função (Move Function)* para passar o comportamento para a classe de dados. Se não puder mover uma função inteira, utilize *Extrair função (Extract Function)* para criar uma função que possa ser movida.

Classes de dados em geral são um sinal de comportamento no lugar errado, o que significa que você poderá fazer grandes progressos passando-o do cliente para a própria classe de dados. Contudo, há exceções, e uma das melhores delas é um registro usado como um registro de resultado para uma chamada de uma função diferente. Um bom exemplo desse caso é a estrutura de dados intermediária após a aplicação de *Separar em fases (Split Phase)*. Uma característica fundamental de um registro de resultados como esse é o fato de ele ser imutável (na prática, pelo menos). Campos imutáveis não precisam ser encapsulados e as informações obtidas de dados imutáveis podem ser representadas como campos em vez de métodos de leitura.

Herança recusada

Subclasses herdam os métodos e os dados de seus pais. Mas e se elas não quiserem ou não precisarem do que lhes for dado? Elas recebem todos esses presentes maravilhosos e escolhem somente alguns para usar.

Segundo a história tradicional, isso quer dizer que a hierarquia está incorreta. Você precisa criar uma nova classe irmã e usar *Descer método (Push Down Method)* e *Descer campo (Push Down Field)* para levar todo o código não utilizado para a classe irmã. Dessa forma, a classe pai armazenará somente o que é comum. Muitas vezes, você ouvirá conselhos dizendo que todas as superclasses devem ser abstratas.

Você perceberá, pelo nosso uso sarcástico de "tradicional", que não daremos esse conselho – nem sempre, ao menos. Usamos subclasses o tempo todo para reutilizar alguns comportamentos, e achamos que essa é uma maneira perfeitamente apropriada de fazer o trabalho. Há um cheiro – e não podemos negá-lo – mas, em geral, não é muito acentuado. Então, dizemos que, se a herança recusada estiver causando confusão e problemas, siga o conselho tradicional. Entretanto, não fique com a impressão de que deve fazer isso o tempo todo. Nove de cada dez vezes, esse odor é muito sutil, a ponto de não valer a pena eliminá-lo.

O cheiro de uma herança recusada será muito mais acentuado se a subclasse estiver reutilizando comportamentos, mas não quiser oferecer suporte à interface da superclasse. Não nos importamos em recusar implementações – mas recusar interface mexe com nossos brios. Nesse caso, porém, não mexa na hierarquia; você deve atacar o problema aplicando *Substituir subclasse por delegação (Replace Subclass with Delegate)* ou *Substituir superclasse por delegação (Replace Superclass with Delegate)*.

Comentários

Não se preocupe, não estamos dizendo que as pessoas não devam escrever comentários. Em nossa analogia olfativa, comentários não são um cheiro ruim; na verdade, são um cheiro agradável. O motivo pelo qual mencionamos os comentários nesta seção é que, com frequência, eles são usados como desodorantes. É surpreendente a frequência com que olhamos um código com muitos comentários e percebemos que eles estão lá porque o código é ruim.

Comentários nos conduzem a um código ruim, com todo tipo de odores desagradáveis que discutimos no restante deste capítulo. Nossa primeira atitude é remover os cheiros ruins por meio da refatoração. Quando

terminamos, muitas vezes percebemos que os comentários eram supérfluos.

Se você precisa de um comentário para explicar o que um bloco de código faz, experimente usar *Extrair função (Extract Function)*. Se o método já tiver sido extraído, mas você ainda precisar de um comentário para explicar o que ele faz, use *Mudar declaração de função (Change Function Declaration)* para renomeá-lo. Se tiver de definir algumas regras sobre o estado necessário ao sistema, use *Introduzir asserção (Introduce Assertion)*.

> *Quando sentir necessidade de escrever um comentário, experimente refatorar o código antes, de modo que qualquer comentário se torne supérfluo.*

Uma boa hora para usar um comentário é quando você não sabe o que fazer. Além de descrever o que está acontecendo, os comentários podem sinalizar áreas sobre as quais você não tem muita certeza. Um comentário também é capaz de explicar por que você fez algo. Esse tipo de informação ajuda quem fará as modificações no futuro, particularmente os que são esquecidos.

CAPÍTULO 4

Escrevendo testes

A refatoração é uma ferramenta importante, mas não pode vir sozinha. Para fazer a refatoração de forma apropriada, preciso ter uma suíte de testes robusta a fim de identificar meus erros inevitáveis. Mesmo com ferramentas de refatoração automatizadas, muitas de minhas refatorações ainda exigirão uma verificação com uma suíte de testes.

Não vejo isso como uma desvantagem. Mesmo sem a refatoração, escrever bons testes aumenta minha eficiência como programador. Foi uma surpresa para mim e é contraintuitivo para a maioria dos programadores – portanto vale a pena explicar por quê.

Importância de um código autotestável

Se você observar como a maioria dos programadores gasta o tempo, verá que escrever código, na verdade, representa uma pequena fração desse tempo. Uma parte é gasta para descobrir o que está acontecendo, outra, no design, mas a maior parte do tempo é gasta com depuração. Tenho certeza de que todo leitor é capaz de se lembrar de ter passado longas horas fazendo depuração – muitas vezes, noite adentro. Todo programador tem uma história de um bug que custou um dia inteiro (ou mais) para ser encontrado. Corrigir o bug em geral é bem rápido, mas encontrá-lo é um pesadelo. Então, quando você corrige um bug, há sempre a chance de que outro apareça e talvez não seja nem sequer notado até ser tarde demais. E você gastará muito tempo para encontrar esse bug.

O evento que me colocou inicialmente no caminho do código autotestável (self-testing code) foi uma palestra no OOPSLA em 1992. Alguém (acho que foi “Bedarra” Dave Thomas) disse, informalmente, que “as classes deveriam conter seus próprios testes”. Então, decidi incorporar testes na base de código, junto com o código de produção. Como eu também estava fazendo um desenvolvimento iterativo, experimentei adicionar testes à medida que terminava cada iteração. O projeto no qual eu estava trabalhando na época era

bem pequeno, de modo que disponibilizávamos iterações a cada semana, ou algo assim. Executar os testes se tornou razoavelmente simples – mas, embora fosse fácil, continuava sendo muito maçante. Isso porque todo teste gerava uma saída no console, a qual eu tinha de conferir. Sou uma pessoa muito preguiçosa e estou disposto a fazer um trabalho bem árduo para evitar mais trabalho. Percebi que, em vez de olhar para a tela e ver se ela havia exibido alguma informação do modelo, eu poderia fazer o computador executar esse teste. Tudo que eu devia fazer era colocar a saída esperada no código de teste e efetuar uma comparação. Eu poderia executar os testes agora, e eles simplesmente exibiriam "OK" na tela se tudo corresse bem. O software passou a ser autotestável.

Garanta que os testes sejam totalmente automatizados e que verifiquem os próprios resultados.

Agora era fácil executar os testes – tão fácil quanto compilar. Então comecei a executar testes sempre que compilava. Logo comecei a notar que minha produtividade havia aumentado drasticamente. Percebi que eu não estava gastando tanto tempo depurando. Se acrescentasse um bug que fosse capturado por um teste anterior, isso se tornaria evidente assim que eu executasse o teste. Esse havia funcionado antes, portanto eu saberia que o bug estava no trabalho feito desde a última vez que eu havia testado. E eu executava os testes com frequência – o que significava que apenas alguns minutos haviam se passado. Então eu sabia que a origem do bug estava no código que eu havia acabado de escrever. Como era uma porção de código pequena, que ainda estava fresca em minha memória, encontrar o bug era fácil. Bugs que, de outra forma, teriam demorado uma hora ou mais para serem encontrados agora exigiam no máximo alguns minutos. Meu software não só era autotestável como, ao executar os testes com frequência, eu tinha um detector de bugs eficaz.

Conforme percebia isso, tornei-me mais agressivo em relação aos testes. Em vez de esperar pelo término de um incremento, eu adicionava os testes logo depois de escrever uma pequena função. Todos os dias eu adicionava algumas funcionalidades novas e os testes para verificá-las. Dificilmente eu gastava mais que alguns minutos procurando um bug de regressão.

Uma suíte de testes é um detector de bugs eficaz, que reduz o tempo necessário para encontrar bugs.

Ferramentas para escrever e organizar esses testes se desenvolveram bastante desde os meus experimentos. Quando voou da Suíça para Atlanta para o OOPSLA em 1997, Kent Beck trabalhou em parceria com Erich Gamma a fim de portar seu framework de testes de unidade, de Smalltalk para Java. O framework resultante, chamado JUnit, tem exercido profunda influência nos testes de programas, servindo de inspiração para uma grande variedade de ferramentas semelhantes [mf-xunit] em muitas linguagens diferentes.

Devo admitir que não é tão fácil persuadir outras pessoas a seguir esse caminho. Escrever os testes significa muito código extra para escrever. A menos que você realmente tenha vivenciado a experiência de como isso agiliza a programação, um código autotestável não parece fazer sentido. O fato de muitas pessoas jamais terem aprendido a escrever testes ou nem mesmo terem pensado neles também não ajudou. Quando feitos manualmente, os testes são extremamente maçantes. Porém, quando são automatizados, pode ser de fato muito divertido escrevê-los.

Sem dúvida, um dos momentos mais convenientes para escrever testes é antes de começar a programar. Quando preciso acrescentar uma funcionalidade, começo escrevendo o teste. Não é uma atitude tão inversa quanto parece. Ao escrever o teste, estou me perguntando o que deve ser feito para acrescentar a funcionalidade. Escrever o teste também faz com que eu me concentre na interface em vez de pensar na implementação (é sempre bom). Também significa que tenho um ponto claro em que termino de escrever o código – é quando o teste funcionar.

Kent Beck transformou esse hábito de escrever os testes antes em uma técnica chamada TDD (Desenvolvimento Orientado a Testes, ou Test-Driven Development) [mf-tdd]. A abordagem de Desenvolvimento Orientado a Testes na programação baseia-se em ciclos rápidos de escrita de um teste (em falha), escrita do código para fazer esse teste funcionar e refatoração para garantir que o resultado seja o mais organizado possível. Esse ciclo de testar-programar-refatorar deve ocorrer muitas vezes por hora, e pode ser uma forma bem produtiva e tranquila de escrever código. Não discutirei mais essa abordagem neste livro, mas a uso, e gentilmente a recomendo.

Chega de polêmica. Embora eu acredite que qualquer pessoa se beneficiaria ao escrever códigos autotestáveis, não é esse o objetivo deste livro. A obra diz respeito à refatoração. A refatoração exige testes. Se quiser refatorar, você deve escrever testes. Este capítulo oferece um ponto de partida para fazer isso

com JavaScript. Este não é um livro sobre testes, portanto não darei muitos detalhes. Percebi, porém, que, com os testes, um pequeno volume de trabalho pode trazer benefícios surpreendentemente grandes.

Como tudo mais neste livro, descreverei a abordagem de testes usando exemplos. Quando desenvolvo um código, escrevo os testes à medida que avanço. Às vezes, porém, preciso refatorar um código que não tem testes – então tenho de deixar o código autotestável antes de começar.

Código de exemplo para testar

Apresentarei um código para ser observado e testado. O código oferece suporte a uma aplicação simples que permite a um usuário analisar e manipular um plano de produção. A UI (crua) tem o seguinte aspecto:

O plano de produção tem uma demanda e o preço para cada província. Cada província tem produtores, e cada um deles é capaz de produzir determinado número de unidades a um preço específico. A UI também mostra a receita que cada produtor teria se vendesse toda a sua produção. Na parte inferior, a tela mostra o déficit (shortfall) na produção (a demanda menos a produção total) e o lucro (profit) para esse plano. A UI permite ao usuário manipular a demanda e o preço, além da produção e custo individuais para ver o efeito no déficit de produção e nos lucros. Sempre que um usuário mudar qualquer número na tela, todos os demais se atualizarão imediatamente.

Estou mostrando uma interface de usuário neste exemplo para que você tenha uma noção de como o software é usado, mas vou me concentrar somente na parte da lógica de negócios do software – isto é, nas classes que calculam o lucro e o déficit, e não no código que gera o HTML e vincula as

mudanças nos campos com a lógica de negócios subjacente. Este capítulo é somente uma introdução ao mundo dos códigos autotestáveis, portanto faz sentido que eu comece com o caso mais simples possível – um código que não envolva interface de usuário, persistência ou interação com serviços externos. Uma separação como essa, porém, é uma boa ideia em qualquer caso: depois que esse tipo de lógica de negócios se torna muito complexo, eu a separo dos mecanismos de UI para que seja possível pensar nela e testá-la mais facilmente.

Esse código da lógica de negócios envolve duas classes: uma que representa um único produtor e outra que representa toda uma província. O construtor da província aceita um objeto JavaScript – que poderíamos supor que seja fornecido por um documento JSON.

Eis o código que carrega a província a partir dos dados JSON:

class Province…

```
constructor(doc) {
  this._name = doc.name;
  this._producers = [];
  this._totalProduction = 0;
  this._demand = doc.demand;
  this._price = doc.price;
  doc.producers.forEach(d => this.addProducer(new Producer(this, d)));
}
addProducer(arg) {
  this._producers.push(arg);
  this._totalProduction += arg.production;
}
```

Essa função cria dados JSON apropriados. Posso criar uma província de exemplo para testes construindo um objeto província com o resultado da função a seguir.

nível mais alto…

```
function sampleProvinceData() {
  return {
    name: "Asia",
    producers: [
      {name: "Byzantium", cost: 10, production: 9},
      {name: "Attalia",   cost: 12, production: 10},
      {name: "Sinope",    cost: 10, production: 6},
    ],
    demand: 30,
```

```
      price: 20
    };
  }
```

A classe de província tem métodos de acesso para os diversos dados:

class Province…

```
  get name() {return this._name;}
  get producers() {return this._producers.slice();}
  get totalProduction() {return this._totalProduction;}
  set totalProduction(arg) {this._totalProduction = arg;}
  get demand() {return this._demand;}
  set demand(arg) {this._demand = parseInt(arg);}
  get price() {return this._price;}
  set price(arg) {this._price = parseInt(arg);}
```

Os setters serão chamados com strings da UI contendo os números, portanto tenho de fazer parse dos números para usá-los de forma confiável nos cálculos.

A classe de produtor é, basicamente, uma classe simples que armazena dados:

class Producer…

```
  constructor(aProvince, data) {
    this._province = aProvince;
    this._cost = data.cost;
    this._name = data.name;
    this._production = data.production || 0;
  }
  get name() {return this._name;}
  get cost() {return this._cost;}
  set cost(arg) {this._cost = parseInt(arg);}

  get production() {return this._production;}
  set production(amountStr) {
    const amount = parseInt(amountStr);
    const newProduction = Number.isNaN(amount) ? 0 : amount;
    this._province.totalProduction += newProduction - this._production;
    this._production = newProduction;
  }
```

O modo como `set production` atualiza o dado derivado na província é feio, e, sempre que vejo isso, quero refatorar para removê-lo. Contudo, tenho de escrever testes antes de refatorar.

O cálculo para o déficit é simples.

class Province…

```
  get shortfall() {
    return this._demand - this.totalProduction;
  }
```

O cálculo para o lucro é um pouco mais complicado.

class Province…

```
  get profit() {
    return this.demandValue - this.demandCost;
  }
  get demandCost() {
    let remainingDemand = this.demand;
    let result = 0;
    this.producers
      .sort((a,b) => a.cost - b.cost)
      .forEach(p => {
        const contribution = Math.min(remainingDemand, p.production);
          remainingDemand -= contribution;
          result += contribution * p.cost;
      });
    return result;
  }
  get demandValue() {
    return this.satisfiedDemand * this.price;
  }
  get satisfiedDemand() {
    return Math.min(this._demand, this.totalProduction);
  }
```

Um teste inicial

Para testar esse código, precisarei de algum tipo de framework de testes. Há vários no mercado, até mesmo exclusivos para JavaScript. Usarei o Mocha [mocha], que é razoavelmente comum e tem boa reputação. Não darei uma explicação completa de como usar o framework, mas mostrarei alguns testes de exemplo com ele. Você deve ser capaz de se adaptar facilmente a um framework diferente para escrever testes parecidos.

Eis um teste simples para o cálculo do déficit:

```
describe('province', function() {
  it('shortfall', function() {
    const asia = new Province(sampleProvinceData());
    assert.equal(asia.shortfall, 5);
```

```
    });
  });
```

O framework Mocha separa o código de teste em blocos, cada qual agrupando uma suíte de testes. Cada teste é colocado em um bloco `it`. Neste caso simples, o teste tem dois passos. O primeiro passo define alguns fixtures – dados e objetos necessários ao teste: nesse caso, um objeto província carregado. A segunda linha verifica algumas característica desse fixture – nesse exemplo, se o déficit é a quantidade esperada, considerando os dados iniciais.

Diferentes desenvolvedores usam as strings descritivas nos blocos `describe` e `it` de modo distinto. Alguns escrevem uma frase que explica o que o teste faz, enquanto outros preferem deixá-los em branco, argumentando que a frase descritiva apenas duplica o código, do mesmo modo que um comentário. Gosto de escrever só o suficiente para identificar qual é o teste quando tenho falhas.

Se executo esse teste em um console NodeJS, a saída terá o seguinte aspecto:

```
,,,,,,,,,,,,,,

  1 passing (61ms)
```

Observe a simplicidade do feedback – somente um resumo de quantos testes foram executados e quantos passaram.

Sempre garanta que um teste falhará quando deve falhar.

Quando escrevo um teste para um código existente como esse, é bom saber que tudo está bem – mas sou naturalmente cético. Em particular, se eu tiver muitos testes executando, sempre fico nervoso achando que um teste não está realmente exercitando o código do modo como acho que deveria e, desse modo, não capturará um bug quando for necessário. Portanto, gosto de ver todo teste falhar no mínimo uma vez quando o escrevo. Meu modo favorito de fazer isso é injetar temporariamente uma falha no código, por exemplo:

class Province…

```
  get shortfall() {
    return this._demand - this.totalProduction * 2;
  }
```

Eis a aparência do console agora:

```
!

  0 passing (72ms)
  1 failing

  1) province shortfall:
     AssertionError: expected -20 to equal 5
      at Context.<anonymous> (src/tester.js:10:12)
```

O framework mostra qual teste falhou e apresenta algumas informações sobre a natureza da falha – nesse caso, qual era o valor esperado e qual o valor que realmente foi gerado. Desse modo, percebo rapidamente que houve uma falha – e posso ver de imediato quais testes falharam, o que me dá uma pista do que deu errado (nesse caso, confirmando que a falha está no lugar em que eu a havia injetado).

> *Execute testes frequentemente. Execute os testes que exercitam o código com o qual você está trabalhando, pelo menos com um intervalo de alguns minutos; execute todos os testes no mínimo uma vez por dia.*

Em um sistema real, posso ter milhares de testes. Um bom framework de testes me permite executá-los de modo fácil e ver rapidamente se algum deles falhou. Esse feedback simples é essencial para um código autotestável. Quando trabalho, executo os testes com bastante frequência – verificando o progresso com um código novo ou conferindo se há erros na refatoração.

O framework Mocha é capaz de usar diferentes bibliotecas, que ele chama de bibliotecas de asserção, a fim de conferir o fixture de um teste. Por ser JavaScript, há zilhões delas por aí, algumas das quais ainda poderão ser atuais quando você estiver lendo este livro. A biblioteca que estou usando no momento se chama Chai [chai]. A Chai permite que eu escreva minhas validações usando um estilo “assert”:

```
describe('province', function() {
  it('shortfall', function() {
    const asia = new Province(sampleProvinceData());
    assert.equal(asia.shortfall, 5);
  });
});
```

ou um estilo “expect”:

```
describe('province', function() {
  it('shortfall', function() {
    const asia = new Province(sampleProvinceData());
```

```
    expect(asia.shortfall).equal(5);
  });
});
```

Em geral, prefiro o estilo assert, mas, no momento, uso mais o estilo expect quando trabalho com JavaScript.

Diferentes ambientes oferecem formas diferentes de executar testes. Quando programo em Java, uso um IDE que tem uma ferramenta gráfica para execução de testes. Sua barra de progresso é verde enquanto todos os testes estiverem passando e se torna vermelha caso algum deles falhe. Meus colegas com frequência usam as expressões “barra verde” e “barra vermelha” para descrever o estado dos testes. Eu poderia dizer, “Nunca refatore com uma barra vermelha”, que significa que você não deve refatorar se sua suíte de testes tiver um teste com falha. Ou eu poderia dizer, “Volte para o verde” para dizer que você deve desfazer alterações recentes e voltar para o último estado em que a suíte de testes estava passando por completo (em geral, voltando para um ponto de verificação [checkpoint] recente no sistema de controle de versões).

Ferramentas gráficas de execução de testes são interessantes, mas não são essenciais. Em geral, configuro meus testes para executar pressionando uma única tecla no Emacs e observo o feedback textual em minha janela de compilação. O ponto principal é que posso ver rapidamente se todos os meus testes estão passando.

Acrescente outro teste

Continuarei acrescentando mais testes agora. Meu estilo é observar tudo que a classe deveria fazer e testar cada um desses aspectos, em quaisquer condições que possam fazer a classe falhar. Não é o mesmo que testar todos os métodos públicos, que é o que alguns programadores defendem. Os testes devem ser orientados a riscos; lembre-se de que estou tentando encontrar bugs – agora ou no futuro. Desse modo, não testo métodos de acesso que simplesmente leiam e escrevam em um campo: eles são tão simples que é pouco provável que eu encontre bugs ali.

Isso é importante porque tentar escrever testes em excesso geralmente resulta em não escrever testes suficientes. Tenho muitos benefícios com os testes, mesmo que eu faça poucos. Meu foco é testar as áreas com as quais me preocupo mais caso veja erros. Desse modo, terei as maiores vantagens possíveis com os esforços investidos em meus testes.

É melhor escrever e executar testes incompletos do que não executar testes completos.

Começarei me concentrando em outra saída importante desse código – o cálculo do lucro. Mais uma vez, escreverei somente um teste básico para o lucro em meu fixture inicial.

```
describe('province', function() {
  it('shortfall', function() {
    const asia = new Province(sampleProvinceData());
    expect(asia.shortfall).equal(5);
  });
  it('profit', function() {
    const asia = new Province(sampleProvinceData());
    expect(asia.profit).equal(230);
  });
});
```

O teste mostra o resultado final, mas eu o obtive definindo antes o valor esperado com um placehoder e então substituindo-o pelo valor gerado pelo programa (230). Eu poderia ter feito esse cálculo manualmente, mas como supõe-se que o código esteja funcionando corretamente, confiarei nele por enquanto. Depois que tiver o novo teste funcionando corretamente, provoco uma falha nele alterando o cálculo do lucro com um `* 2` espúrio. Satisfaço a mim mesmo vendo que o teste falha conforme deveria, e depois removo a falha injetada. Esse padrão – escrever com um placeholder para o valor esperado, substituir o placeholder pelo valor do código, injetar uma falha, removê-la – é um padrão comum, que uso quando acrescento testes em um código existente.

Há um pouco de duplicação entre esses testes – ambos definem o fixture com a mesma linha inicial. Do mesmo modo que suspeito quando há código duplicado em um código comum, suspeito dele no código de testes, portanto procurarei removê-lo fatorando para um lugar comum. Uma opção é levar a constante para o escopo mais externo.

```
describe('province', function() {
  const asia = new Province(sampleProvinceData()); // NÃO FAÇA ISSO
  it('shortfall', function() {
    expect(asia.shortfall).equal(5);
  });
  it('profit', function() {
    expect(asia.profit).equal(230);
```

```
    });
  });
```

Contudo, como mostra o comentário, nunca faço isso. Funcionará no momento, mas introduzirei uma placa de Petri preparada para um dos piores bugs de teste – um fixture compartilhado que fará os testes interagirem. A palavra reservada const em JavaScript significa que apenas a referência para asia é constante, e não o conteúdo desse objeto. Caso um teste futuro altere esse objeto comum, acabarei com falhas intermitentes porque os testes interagirão por meio do fixture compartilhado, gerando resultados diferentes conforme a ordem em que os testes são executados. Os testes não são determinísticos, e isso pode resultar, no melhor caso, em uma depuração difícil e demorada, e, no pior caso, em quebra de confiança nos testes. Em vez disso, prefiro fazer o seguinte:

```
describe('province', function() {
  let asia;
  beforeEach(function() {
    asia = new Province(sampleProvinceData());
  });
  it('shortfall', function() {
    expect(asia.shortfall).equal(5);
  });
  it('profit', function() {
    expect(asia.profit).equal(230);
  });
});
```

A cláusula beforeEach é executada antes de cada teste, limpando asia e definindo-a com um novo valor a cada execução. Com isso, tenho um novo fixture antes de cada teste executar, mantendo os testes isolados e evitando uma falta de determinismo causador de muitos problemas.

Quando dou esse conselho, algumas pessoas se preocupam com o fato de que definir um novo fixture a cada execução deixe os testes mais lentos. Na maioria das vezes, isso não será perceptível. Se for um problema, eu consideraria ter um fixture compartilhado; nesse caso, porém, terei de ser realmente cuidadoso e garantir que nenhum teste vai alterá-lo. Também posso usar um fixture compartilhado se tiver certeza de que ele é imutável. Minha reação imediata, porém, é usar um fixture novo porque já sofri demais no passado com o custo de depuração por um erro cometido em consequência de um fixture compartilhado.

Considerando que executo o código de configuração em beforeEach a cada

teste, por que não deixar esse código nos blocos it individuais? Gosto quando todos os meus testes atuam em um fixture comum, de modo que posso me familiarizar com esse fixture padrão e observar as várias características testadas com ele. A presença do bloco beforeEach sinaliza ao leitor que estou usando um fixture padrão. Você pode então ver todos os testes no escopo desse bloco describe e saberá que todos usam os mesmos dados básicos como ponto de partida.

Modificando o fixture

Até agora, os testes que escrevi mostram como faço uma sondagem das propriedades do fixture depois de tê-lo carregado. Quando em uso, porém, esse fixture será constantemente atualizado pelos usuários à medida que eles alterarem os valores.

A maioria das atualizações são setters simples e, em geral, não me preocupo em testá-los, pois há poucas chances de serem fonte de bugs. No entanto, há um comportamento complicado em torno do setter de production de Producer, portanto acho que vale a pena escrever um teste.

describe('province'…

```
  it('change production', function() {
      asia.producers[0].production = 20;
      expect(asia.shortfall).equal(-6);
      expect(asia.profit).equal(292);
  });
```

Esse é um padrão comum. Tomo o fixture padrão inicial configurado no bloco beforeEach, exercito esse fixture no teste, e então verifico se o fixture fez o que acho que ele deveria ter feito. Se você já leu bastante sobre testes, verá que essas fases são descritas de forma variada como setup-exercise-verify (configurar-exercitar-verificar), given-when-then (dado-quando-então) ou arrange-act-assert (organizar-atuar-garantir). Às vezes, você verá todos os passos presentes no próprio teste; em outros casos, as primeiras fases comuns poderão ser colocadas em rotinas de configuração padrão, como em beforeEach.

(Há uma quarta fase implícita que, em geral, não é mencionada: teardown (desmontagem). A fase de teardown remove o fixture entre os testes, de modo que não haja interação entre diferentes testes. Ao fazer toda a minha configuração em beforeEach, permito que o framework de testes desmonte implicitamente meu fixture entre os testes, portanto tenho a fase de

teardown gratuitamente. A maioria dos autores de testes ignora a fase de teardown – e é razoável que seja assim, pois, na maioria das vezes, nós a ignoramos. Às vezes, porém, pode ser importante ter uma operação explícita de teardown, particularmente se houver um fixture que seja necessário compartilhar entre os testes por ser demorado para criar.)

Nesse teste, estou verificando duas características diferentes em uma única cláusula `it`. Como regra geral, ter somente uma única instrução de verificação em cada cláusula `it` é uma atitude sábia. Isso porque o teste falhará na primeira verificação de falha – muitas vezes, esse fato ocultará informações úteis quando você estiver tentando descobrir por que um teste falhou. Nesse caso, considero que os dois testes estão intimamente conectados, a ponto de ficar satisfeito em tê-los no mesmo teste. Se eu quiser separá-los em cláusulas `it` distintas, poderei fazer isso mais tarde.

Sondando os limites

Até agora, meus testes tiveram como foco o uso comum, em geral chamado de condições do "caminho feliz" (happy path), em que tudo está bem e é usado conforme esperado. Contudo, também é bom fazer testes nos limites dessas condições – para ver o que acontecerá se algo puder dar errado.

Sempre que tenho uma coleção de itens, por exemplo, os produtores neste exemplo, gosto de ver o que acontecerá se ela estiver vazia.

```
describe('no producers', function() {
  let noProducers;
  beforeEach(function() {
    const data = {
      name: "No proudcers",
      producers: [],
      demand: 30,
      price: 20
    };
    noProducers = new Province(data);
  });
  it('shortfall', function() {
    expect(noProducers.shortfall).equal(30);
  });
  it('profit', function() {
    expect(noProducers.profit).equal(0);
  });
```

Com números, os zeros são um bom aspecto a ser sondado:

describe('province'…

```
  it('zero demand', function() {
    asia.demand = 0;
      expect(asia.shortfall).equal(-25);
      expect(asia.profit).equal(0);
  });
```

assim como os negativos:

describe('province'…

```
  it('negative demand', function() {
    asia.demand = -1;
    expect(asia.shortfall).equal(-26);
    expect(asia.profit).equal(-10);
  });
```

A essa altura, posso começar a me perguntar se uma demanda negativa resultando em um lucro negativo realmente faz sentido no domínio. A demanda mínima não deveria ser zero? Nesse caso, o setter talvez devesse reagir de modo diferente a um argumento negativo – gerando um erro ou definindo o valor com zero. São boas perguntas para se fazer, e escrever testes como esses me ajuda a pensar em como o código deve reagir em casos limites.

Pense nas condições limites nas quais algo pode dar errado e concentre aí seus testes.

Os setters aceitam uma string proveniente dos campos da UI, que estão limitados a aceitar somente números – os campos, porém, ainda podem estar em branco, portanto devo ter testes que garantam que o código responderá aos brancos conforme esperado.

describe('province'…

```
  it('empty string demand', function() {
    asia.demand = "";
    expect(asia.shortfall).NaN;
    expect(asia.profit).NaN;
  });
```

Observe como desempenho o papel de um inimigo para o meu código. Estou pensando ativamente em como posso causar uma falha nele. Acho esse tipo de raciocínio, ao mesmo tempo, produtivo e divertido. Ele me permite usar a

parte malvada de minha psique.

O caso a seguir é interessante:

```
describe('string for producers', function() {
  it('', function() {
    const data = {
      name: "String producers",
      producers: "",
      demand: 30,
      price: 20
    };
    const prov = new Province(data);
    expect(prov.shortfall).equal(0);
  });
```

Esse teste não gera uma falha simples informando que o déficit não é 0. Eis a saída do console:

```
,,,,,,,,,!
  9 passing (74ms)
  1 failing

  1) string for producers :
     TypeError: doc.producers.forEach is not a function
      at new Province (src/main.js:22:19)
      at Context.<anonymous> (src/tester.js:86:18)
```

O Mocha trata isso como uma falha – mas muitos frameworks de teste fazem uma distinção entre essa situação, que eles chamam de erro, e uma falha comum. Uma falha sinaliza um passo de verificação em que o valor propriamente dito está fora dos limites esperados pela instrução de verificação. Esse erro, porém, é de outra natureza – é uma exceção gerada durante uma fase anterior (nesse caso, na configuração). Parece ser uma exceção não prevista pelos autores do código, de modo que vemos, lamentavelmente, um erro conhecido pelos programadores de JavaScript ("… is not a function", isto é, "… não é uma função").

Como o código deveria responder em um caso como esse? Uma abordagem seria acrescentar um tratamento que apresentasse uma resposta de erro mais apropriada – gerando uma mensagem de erro mais significativa ou simplesmente definindo `producers` com um array vazio (adicionando uma mensagem de log, talvez). Mas pode haver motivos válidos para deixar o código como está. Talvez o objeto de entrada seja gerado por uma fonte confiável – por exemplo, por outra parte da mesma base de código. Inserir

muitas verificações de validação entre módulos da mesma base de código pode resultar em verificações duplicadas que causem mais problemas, especialmente se duplicarem validações feitas em outros lugares. No entanto, se esse objeto de entrada tiver origem externa, por exemplo, uma requisição codificada em JSON, as verificações de validação serão necessárias, e deverão ser testadas. Qualquer que seja o caso, escrever testes como esse levanta esse tipo de questões.

Se eu estivesse escrevendo testes como esse antes de refatorar, provavelmente o descartaria. A refatoração deve preservar o comportamento observável; um erro como esse está fora dos limites do observável, portanto não preciso me preocupar com o fato de minha refatoração alterar a resposta do código a essa condição.

Se esse erro puder levar a dados ruins no programa, causando uma falha que será difícil de depurar, posso usar *Introduzir asserção (Introduce Assertion)* para que o teste falhe mais rápido. Não acrescento testes para capturar essas falhas de asserção porque elas são, por si só, uma forma de teste.

*Não deixe que o medo de os testes não capturarem **todos** os bugs impeça você de escrever testes que capturem a **maioria** dos bugs.*

Quando você deve parar? Tenho certeza de que você já deve ter ouvido muitas vezes que é impossível provar por meio de testes que um programa não tenha bugs. Isso é verdade, mas não afeta a capacidade de os testes deixarem a programação mais rápida. Já vi muitas regras propostas para garantir que você testou todas as combinações de tudo. Vale a pena dar uma olhada nelas – mas não se deixe dominar por elas. Há uma lei de diminuição de retornos nos testes, e há o perigo de que, ao tentar escrever testes demais, você se sinta desencorajado e acabe não escrevendo nenhum. Concentre-se nos pontos em que estão os riscos. Observe o código e veja os lugares em que ele se torna complexo. Olhe para uma função e considere as prováveis áreas em que pode haver erros. Seus testes não encontrarão todos os bugs, mas, à medida que refatorar, você compreenderá melhor o programa e, desse modo, encontrará mais bugs. Embora eu sempre comece refatorando com uma suíte de testes, invariavelmente faço acréscimos a ela à medida que prossigo.

Muito além disso

Esse é o máximo que avançarei neste capítulo – afinal de contas, este não é

um livro sobre testes, mas sobre refatoração. Testes, porém, são um assunto importante, tanto porque são uma base necessária para a refatoração como por serem uma ferramenta importante por si só. Embora eu esteja feliz em ver o crescimento da refatoração como uma prática de programação desde que escrevi este livro, estou mais feliz ainda em ver a mudança de atitude em relação aos testes. Anteriormente vistos como responsabilidade de um grupo separado (e inferior), hoje em dia os testes são uma preocupação cada vez mais importante para qualquer desenvolvedor de software que se preze. Com frequência, as arquiteturas são julgadas, corretamente, com base na testabilidade.

Os tipos de testes que mostrei neste capítulo são testes de unidades, projetados para atuar em uma pequena área de código e de forma rápida. Eles são a espinha dorsal do código autotestável; a maioria dos testes em um sistema como esse são testes de unidade. Há outros tipos de testes também, com foco na integração entre componentes, exercitando vários níveis de software ao mesmo tempo, procurando problemas de desempenho etc. (Muito mais variados que os tipos de teste são as discussões que as pessoas travam sobre como classificar os testes.)

Assim como a maioria dos aspectos da programação, os testes são uma atividade iterativa. A menos que você seja muito habilidoso ou muito sortudo, seus testes não estarão corretos na primeira tentativa. Eu noto que trabalho constantemente na suíte de testes – tanto quanto trabalho no código principal. Naturalmente isso quer dizer adicionar novos testes à medida que acrescento novas funcionalidades, mas envolve também observar os testes existentes. Eles são suficientemente claros? Preciso refatorá-los para que seja mais fácil entender o que fazem? Tenho os testes corretos? Um hábito importante a ser desenvolvido é responder a um bug escrevendo primeiro um teste que claramente o evidencie. Somente depois de ter o teste é que devo corrigir o bug. Ao ter o teste, sei que o bug permanecerá inativo. Também penso no bug e em seu teste: ele me dá pistas sobre outras lacunas na suíte de testes?

Ao receber o relatório de um bug, comece escrevendo um teste de unidade que exponha o bug.

Uma pergunta comum é “Quantos testes são suficientes?”. Não há uma boa medida para isso. Algumas pessoas defendem o uso da cobertura de testes [mf-tc] como uma medida, mas uma análise de cobertura dos testes é boa

somente para identificar áreas de código não testadas, e não para avaliar a qualidade de uma suíte de testes.

A melhor medida para saber se uma suíte de testes é boa o suficiente é subjetiva: qual é o seu nível de confiança de que, caso alguém introduza um defeito no código, algum teste falhará? Isso não é uma medida que possa ser analisada de forma objetiva, e não leva em conta uma falsa confiança, mas o objetivo de um código autotestável é adquirir essa confiança. Se posso refatorar meu código e estar bem certo de que não introduzi nenhum bug porque meus testes deram sinal verde, então posso me considerar satisfeito por ter testes suficientemente bons.

É possível escrever testes demais. Um sinal disso é quando gasto mais tempo alterando os testes do que o código em teste – e sinto que os testes estão causando lentidão. Porém, embora haja casos de testes em excesso, é extremamente raro acontecer, em comparação com os casos em que há falta de testes.

CAPÍTULO 5

Apresentação do catálogo

O restante deste livro contém um catálogo de refatorações. Esse catálogo nasceu de anotações pessoais que fiz para me lembrar de como fazer as refatorações de modo seguro e eficaz. Desde então, venho detalhando o catálogo, com mais informações provenientes de uma exploração deliberada de alguns passos da refatoração. É algo que ainda uso quando faço uma refatoração que não utilizo há um bom tempo.

Formato das refatorações

Para descrever as refatorações no catálogo, uso um formato padrão. Cada refatoração tem cinco partes, que são as seguintes:

- Começo com um nome. O nome é importante para construir um vocabulário das refatorações. Esse é o nome que utilizo nos demais lugares no livro. As refatorações muitas vezes usam nomes diferentes hoje em dia, portanto listei também quaisquer aliases que pareçam comuns.
- Após o nome, há um pequeno esboço da refatoração. Ele ajuda a encontrar rapidamente uma refatoração.
- A motivação descreve por que a refatoração deve ser feita e as circunstâncias em que ela não deve ser feita.
- O procedimento é uma descrição concisa, passo a passo, de como fazer a refatoração.
- Os exemplos mostram um uso bem simples da refatoração para demonstrar como ela funciona.

O esboço mostra um exemplo de código da transformação feita pela refatoração. Não foi feito para explicar o que é a refatoração, muito menos para mostrar como fazê-la, mas deve fazer você se lembrar de como ela é, caso já tenha deparado com ela antes. Do contrário, você provavelmente terá de entender o exemplo para ter uma ideia melhor. Incluo também um

pequeno diagrama; novamente, não tenho a intenção de que ele seja usado para explicações – o diagrama serve mais para ajudar a memorizar.

O procedimento nasceu de minhas próprias anotações, para eu me lembrar de como fazer a refatoração caso não a tenha usado há um tempo. Desse modo, ele é, de certo modo, conciso, geralmente sem explicações dos motivos pelos quais os passos são executados daquela maneira. Dou uma explicação mais detalhada no exemplo. O procedimento é composto de anotações breves que você pode referenciar facilmente caso conheça a refatoração, mas tenha de consultar os passos (pelo menos, é assim que eu o uso). Provavelmente você precisará ler os exemplos quando fizer a refatoração pela primeira vez.

Descrevi o procedimento de modo que cada passo das refatorações fosse o menor possível. Dou ênfase à forma segura de fazer a refatoração: dar passos bem pequenos e testar depois de cada passo. Quando trabalho, em geral dou passos maiores do que alguns dos passinhos de bebê descritos, mas, se eu deparar com um bug, desfaço o último passo e executo os passos menores. Os passos incluem uma série de referências para casos especiais. Desse modo, eles também funcionam como uma lista de verificação; eu mesmo me esqueço desses casos com frequência.

Embora eu (com poucas exceções) liste somente um conjunto de passos para o procedimento, essa não é a única maneira de fazer a refatoração. Selecionei os procedimentos do livro porque, na maioria das vezes, eles funcionam muito bem. É provável que você os varie à medida que adquirir mais prática com a refatoração, e isso não será um problema. Basta lembrar que o segredo é dar passos pequenos – e, quanto mais complicada a situação, menores devem ser os passos.

Os exemplos são do tipo ridiculamente simples dos livros didáticos. Meu objetivo com eles é ajudar a explicar a refatoração básica com um mínimo de distrações, portanto espero que você me perdoe a simplicidade. (Eles certamente não são exemplos de uma boa modelagem de negócios.) Tenho certeza de que você será capaz de aplicá-los em suas situações muito mais complexas. Algumas refatorações muito simples não têm exemplos porque não achei que um exemplo acrescentasse muito.

Em particular, lembre-se de que os exemplos foram incluídos somente para demonstrar a refatoração específica em discussão. Na maioria dos casos, ainda haverá problemas com o código final – todavia, corrigir esses problemas exige outras refatorações. Em alguns casos em que as refatorações

em geral ocorrem juntas, levo os exemplos de uma refatoração para outra. Na maioria das vezes, mantenho o código como está após a refatoração única. Faço isso para deixar cada refatoração autocontida, pois a principal função do catálogo é servir de referência.

Uso cores para dar destaque ao código alterado quando for difícil identificá-lo no meio do código que não foi modificado. Não destaco todos os códigos alterados porque um uso excessivo fará com que o propósito seja comprometido.

Escolha das refatorações

Este não é, de forma alguma, um catálogo completo das refatorações. Ele é – espero – um conjunto das refatorações mais úteis que merecem ser descritas por escrito. Por "mais úteis" quero dizer aquelas que são mais comuns e que valem a pena ser nomeadas e descritas. Considero que vale a pena descrever uma refatoração de acordo com uma combinação de motivos: algumas refatorações têm procedimentos interessantes que ajudam a desenvolver habilidades gerais de refatoração, enquanto outras têm um grande efeito no aperfeiçoamento do design do código.

Algumas refatorações estão ausentes porque são tão pequenas e simples que não achei que valesse a pena descrevê-las. Um exemplo na primeira edição era *Deslocar instruções (Slide Statements)* – que uso frequentemente, mas não havia reconhecido como uma refatoração que devesse ser incluída no catálogo (obviamente mudei de ideia nesta edição). Algumas podem muito bem ser acrescentadas no livro com o tempo, dependendo da quantidade de energia que eu dedicar a novas refatorações no futuro.

Outra categoria de refatorações são aquelas que existem logicamente, mas não são muito usadas por mim ou que exibem uma semelhança simples com outras refatorações. Toda refatoração neste livro tem uma refatoração lógica inversa, mas não descrevi todas porque não considero muitas delas interessantes. *Encapsular variável (Encapsulate Variable)* é uma refatoração comum e eficaz, mas raramente uso a sua inversa (de qualquer modo, é fácil fazê-la), portanto não achei que precisássemos de uma entrada no catálogo para ela.

CAPÍTULO 6

Primeiro conjunto de refatorações

Inicio o catálogo com um conjunto de refatorações que considero mais úteis conhecer primeiro.

Provavelmente, a refatoração mais comum que faço é extrair código para uma função – *Extrair função (Extract Function)* – ou extrair uma variável – *Extrair variável (Extract Variable)*. Como a refatoração tem tudo a ver com mudanças, não é nenhuma surpresa que eu também use com frequência as inversas dessas duas refatorações: *Internalizar função (Inline Function)* e *Internalizar variável (Inline Variable)*.

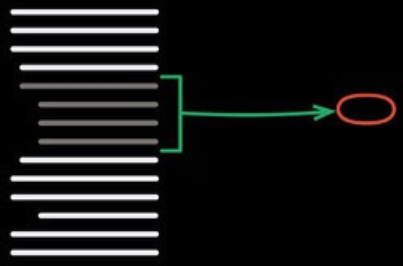
A extração diz respeito a dar nomes e, muitas vezes, é necessário modificar os nomes à medida que vou aprendendo. *Mudar declaração de função (Change Function Declaration)* muda os nomes de funções; também uso essa refatoração para adicionar ou remover argumentos de uma função. Para variáveis, uso *Renomear variável (Rename Variable)*, que conta com o uso de *Encapsular variável (Encapsulate Variable)*. Ao alterar argumentos de funções, muitas vezes acho conveniente combinar um grupo comum de argumentos em um único objeto usando *Introduzir objeto de parâmetros (Introduce Parameter Object)*.

Criar e nomear funções são refatorações de baixo nível essenciais – mas, depois de criadas, é necessário agrupar as funções em módulos de nível mais alto. Uso *Combinar funções em classe (Combine Functions into Class)* para agrupar funções, junto com os dados nos quais elas atuam, em uma classe. Outra alternativa disponível é combiná-las em uma transformação – *Combinar funções em transformação (Combine Functions into Transform)* –, que é particularmente conveniente para dados somente de leitura. Dando um passo além na escala, muitas vezes posso fazer uma composição desses módulos em fases distintas de processamento usando *Separar em fases (Split*

Phase).

Extrair função (Extract Function)

anteriormente: *Extrair método* (Extract Method)
inversa de: *Internalizar função (Inline Function)*



```
function printOwing(invoice) {
  printBanner();
  let outstanding =
calculateOutstanding();
  // exibe detalhes
  console.log(`name:
${invoice.customer}`);
  console.log(`amount: ${outstanding}`);
}
```

```
function printOwing(invoice) {
  printBanner();
  let outstanding = calculateOutstanding();
  printDetails(outstanding);
  function printDetails(outstanding) {
    console.log(`name:
${invoice.customer}`);
    console.log(`amount: ${outstanding}`);
  }
}
```

Motivação

Extrair função é uma das refatorações mais comuns que faço. (Nesse caso, uso o termo “função”, mas o mesmo vale para um método em uma linguagem orientada a objetos, ou para qualquer tipo de procedimento ou sub-rotina.)

Observo um fragmento de código, entendo o que ele faz e, em seguida, extraio esse código para a sua própria função, com um nome baseado em seu propósito.

Durante minha carreira, já ouvi vários argumentos sobre quando colocar um código em sua própria função. Algumas dessas diretrizes baseavam-se no tamanho: funções não devem ser maiores que o tamanho de uma tela. Outras eram baseadas em reutilização: qualquer código usado mais de uma vez deve ser colocado em sua própria função, mas um código utilizado somente uma vez deve permanecer interno. O argumento que mais faz sentido para mim, no entanto, é a separação entre intenção e implementação. Se você tiver de dispender esforços para observar um fragmento de código e descobrir *o que* ele faz, então deve extraí-lo em uma função e atribuir-lhe um nome com base no "o quê". Então, quando o ler novamente, o propósito da função estará evidente, e, na maioria das vezes, você não terá de se preocupar sobre como a função atende ao seu propósito (que está no corpo da função).

Após ter aceitado esse princípio, desenvolvi o hábito de escrever funções muito pequenas – tipicamente, com apenas algumas linhas. Para mim, qualquer função com mais de meia dúzia de linhas de código começa a exalar um cheiro, e não é incomum que eu tenha funções com uma única linha de código. O fato de o tamanho não ser importante se tornou claro para mim por causa de um exemplo do sistema Smalltalk original que Kent Beck me mostrou. Naquela época, Smalltalk executava em sistemas preto e branco. Se você quisesse destacar um texto ou uma imagem, era necessário usar o vídeo reverso. A classe gráfica de Smalltalk tinha um método para isso, chamado `highlight`, cuja implementação era apenas uma chamada ao método `reverse`. O nome do método era mais extenso que sua implementação – mas isso não importava porque havia uma distância enorme entre a intenção do código e a sua implementação.

Algumas pessoas se preocupam com funções pequenas porque ficam apreensivas com o custo de uma chamada de função no que concerne ao desempenho. Quando eu era jovem, esse fato ocasionalmente era relevante, mas nos dias de hoje é muito raro. Compiladores que fazem otimizações muitas vezes trabalham melhor com funções menores, que podem ser colocadas em cache mais facilmente. Como sempre, siga as diretrizes gerais para otimização com vistas ao desempenho.

Funções pequenas como essa só funcionam se os nomes forem bons, portanto você deve prestar bastante atenção na nomenclatura. Isso exige

prática – mas, depois que você se torna bom nisso, essa abordagem pode fazer com que o código se torne notadamente a própria documentação.

Muitas vezes, vejo fragmentos de código em uma função maior que começam com um comentário para dizer o que fazem. O comentário geralmente é uma boa dica para o nome da função quando extraio esse fragmento.

Procedimento

- Crie uma nova função e dê-lhe um nome de acordo com o seu propósito (nomeie a função com base no que ela faz, e não em como ela faz).

Se o código que quero extrair for muito simples, por exemplo, é uma única chamada de função, mesmo assim, extrairei o código se o nome da nova função revelar o propósito do código de modo mais apropriado. Se eu não conseguir pensar em um nome mais significativo, é sinal de que eu não deveria extrair o código. No entanto, não preciso pensar no melhor nome possível de imediato; às vezes, um bom nome só surgirá durante a extração. Não há problemas em extrair uma função, tentar trabalhar com ela, perceber que não está ajudando e, então, internalizá-la novamente. Desde que eu tenha aprendido algo, meu tempo não terá sido desperdiçado.

Se a linguagem aceitar funções aninhadas, deixe a função extraída aninhada na função original. Isso reduzirá a quantidade de variáveis fora do escopo com as quais será preciso lidar depois dos próximos passos. É sempre possível usar *Mover função (Move Function)* mais tarde.

- Copie o código extraído da função original para a nova função final.
- Verifique o código extraído em busca de referências a qualquer variável que seja local no escopo da função original, e que não estará no escopo da função extraída. Passe-as como parâmetros.

Se eu fizer a extração do código para uma função aninhada dentro da função original, não terei esses problemas.

Em geral, são variáveis locais e parâmetros da função. A abordagem mais genérica é passar todos esses parâmetros como argumentos. Geralmente não haverá dificuldades para variáveis usadas, mas que não recebem valor.

Se uma variável for usada apenas no código extraído, mas estiver declarada fora dele, mova a declaração para o código extraído.

Qualquer variável que tenha algum valor atribuído exigirá mais atenção se

for passada por valor. Se houver somente uma delas, tento tratar o código extraído como uma consulta e atribuir o resultado à variável em questão.

Às vezes, percebo que há muitas variáveis locais recebendo valor no código extraído. É melhor abandonar a extração nesse ponto. Se isso acontecer, considero outras refatorações como *Separar variável (Split Variable)* ou *Substituir variável temporária por consulta (Replace Temp with Query)* para simplificar o uso das variáveis e retomo a extração mais tarde.

- Compile depois que tiver cuidado de todas as variáveis.

Depois de ter tratado todas as variáveis, pode ser conveniente compilar se o ambiente da linguagem fizer verificações em tempo de compilação. Muitas vezes isso ajudará a identificar qualquer variável que não tenha sido tratada adequadamente.

- Substitua o código extraído da função original por uma chamada da função final.
- Teste.
- Procure outros códigos iguais ou parecidos com o código que acabou de ser extraído e considere o uso de *Substituir código internalizado por chamada de função (Replace Inline Code with Function Call)* para chamar a nova função.

Algumas ferramentas de refatoração têm suporte direto para isso. Caso contrário, pode valer a pena fazer buscas rápidas para ver se há códigos duplicados em outros lugares.

Exemplo: sem variáveis fora do escopo

No caso mais simples, *Extrair função* é trivial.

```
function printOwing(invoice) {
  let outstanding = 0;

  console.log("**********************");
  console.log("**** Customer Owes ****");
  console.log("**********************");

  // calcula o valor a pagar (outstanding)
  for (const o of invoice.orders) {
    outstanding += o.amount;
  }

  // registra a data de vencimento (due date)
```

```
  const today = Clock.today;
  invoice.dueDate = new Date(today.getFullYear(),
    today.getMonth(), today.getDate() + 30);

  // exibe detalhes
  console.log(`name: ${invoice.customer}`);
  console.log(`amount: ${outstanding}`);
  console.log(`due: ${invoice.dueDate.toLocaleDateString()}`);
}
```

Você pode estar se perguntando de que se trata `Clock.today`. *É um Clock Wrapper (Encapsulador de relógio) [mf-cw] – um objeto que encapsula chamadas para o relógio do sistema. Evito fazer chamadas diretas como* `Date.now()` *em meu código, pois isso resulta em testes não determinísticos e dificulta reproduzir as condições de erro ao diagnosticar falhas.*

É fácil extrair o código que exibe o banner. Basta cortar, colar e fazer uma chamada:

```
function printOwing(invoice) {
  let outstanding = 0;

  printBanner();

  // calcula o valor a pagar (outstanding)
  for (const o of invoice.orders) {
    outstanding += o.amount;
  }

  // registra a data de vencimento (due date)
  const today = Clock.today;
  invoice.dueDate = new Date(today.getFullYear(), today.getMonth(), today.getDate() + 30);

  // exibe detalhes
  console.log(`name: ${invoice.customer}`);
  console.log(`amount: ${outstanding}`);
  console.log(`due: ${invoice.dueDate.toLocaleDateString()}`);
}
function printBanner() {
  console.log("**************************");
  console.log("**** Customer Owes ****");
  console.log("**************************");
}
```

De modo semelhante, posso extrair também o código para exibição dos detalhes:

```
function printOwing(invoice) {
```

```
  let outstanding = 0;

  printBanner();

  // calcula o valor a pagar (outstanding)
  for (const o of invoice.orders) {
    outstanding += o.amount;
  }

  // registra a data de vencimento (due date)
  const today = Clock.today;
  invoice.dueDate = new Date(today.getFullYear(), today.getMonth(), today.getDate() + 30);

  printDetails();

  function printDetails() {
    console.log(`name: ${invoice.customer}`);
    console.log(`amount: ${outstanding}`);
    console.log(`due: ${invoice.dueDate.toLocaleDateString()}`);
  }
```

Isso faz com que *Extrair função* pareça uma refatoração extremamente trivial. Contudo, em muitas situações, ela acaba sendo mais complicada.

No exemplo anterior, defini `printDetails` de modo que ficasse aninhada em `printOwing`. Dessa forma, ela é capaz de acessar todas as variáveis definidas em `printOwing`. No entanto, essa não será uma opção para mim se eu estiver programando em uma linguagem que não permita funções aninhadas. Nesse caso, eu estaria essencialmente diante do problema de extrair a função para o nível mais alto; isso significa que terei de prestar atenção em qualquer variável que exista apenas no escopo da função original. São os argumentos da função original e as variáveis temporárias definidas na função.

Exemplo: usando variáveis locais

O caso mais simples com variáveis locais é aquele em que elas são usadas, mas não recebem novos valores. Nessa situação, posso simplesmente as passar como parâmetros. Desse modo, se eu tiver a função a seguir:

```
function printOwing(invoice) {
  let outstanding = 0;

  printBanner();

  // calcula o valor a pagar (outstanding)
  for (const o of invoice.orders) {
    outstanding += o.amount;
```

```
    }

    // registra a data de vencimento (due date)
    const today = Clock.today;
    invoice.dueDate = new Date(today.getFullYear(), today.getMonth(), today.getDate() + 30);

    // exibe detalhes
    console.log(`name: ${invoice.customer}`);
    console.log(`amount: ${outstanding}`);
    console.log(`due: ${invoice.dueDate.toLocaleDateString()}`);
  }
```

Posso extrair a exibição dos detalhes passando dois parâmetros:

```
  function printOwing(invoice) {
    let outstanding = 0;

    printBanner();

    // calcula o valor a pagar (outstanding)
    for (const o of invoice.orders) {
      outstanding += o.amount;
    }

    // registra a data de vencimento (due date)
    const today = Clock.today;
    invoice.dueDate = new Date(today.getFullYear(), today.getMonth(), today.getDate() + 30);

    printDetails(invoice, outstanding);
  }
  function printDetails(invoice, outstanding) {
    console.log(`name: ${invoice.customer}`);
    console.log(`amount: ${outstanding}`);
    console.log(`due: ${invoice.dueDate.toLocaleDateString()}`);
  }
```

O mesmo vale se a variável local for uma estrutura (por exemplo, um array, um registro ou um objeto) e eu modificar essa estrutura. Assim, de modo semelhante, posso extrair a definição da data de vencimento (due date):

```
  function printOwing(invoice) {
    let outstanding = 0;

    printBanner();

    // calcula o valor a pagar (outstanding)
    for (const o of invoice.orders) {
      outstanding += o.amount;
    }
```

```
  recordDueDate(invoice);
  printDetails(invoice, outstanding);
}
function recordDueDate(invoice) {
  const today = Clock.today;
  invoice.dueDate = new Date(today.getFullYear(), today.getMonth(), today.getDate() + 30);
}
```

Exemplo: atribuindo novo valor a uma variável local

É a atribuição a variáveis locais que se torna complicada. Nesse caso, estamos falando apenas de variáveis temporárias. Se eu vir uma atribuição a um parâmetro, uso imediatamente *Separar variável (Split Variable)*, transformando-o em uma variável temporária.

Para variáveis temporárias que recebam valor, há dois casos. O mais simples é aquele em que a variável é temporária e usada somente dentro do código extraído. Se isso acontecer, a variável existirá apenas no código extraído. Às vezes, particularmente quando as variáveis são inicializadas um pouco distantes do local em que são usadas, será conveniente usar *Deslocar instruções (Slide Statements)* para deixar todas as manipulações da variável juntas.

O caso mais complicado é aquele em que se usa a variável fora da função extraída. Nesse caso, tenho de devolver o novo valor. Posso ilustrar isso com a função a seguir, de aspecto familiar:

```
function printOwing(invoice) {
  let outstanding = 0;

  printBanner();

  // calcula o valor a pagar (outstanding)
  for (const o of invoice.orders) {
    outstanding += o.amount;
  }

  recordDueDate(invoice);
  printDetails(invoice, outstanding);
}
```

Mostrei as refatorações anteriores executando tudo em um só passo, pois eram simples; agora, porém, darei um passo de cada vez, de acordo com o procedimento.

Em primeiro lugar, deslocarei a declaração para que fique perto de seu uso.

```
function printOwing(invoice) {
  printBanner();

  // calcula o valor a pagar (outstanding)
  let outstanding = 0;
  for (const o of invoice.orders) {
    outstanding += o.amount;
  }

  recordDueDate(invoice);
  printDetails(invoice, outstanding);
}
```

Então, copio o código que quero extrair para a função final.

```
function printOwing(invoice) {
  printBanner();

  // calcula o valor a pagar (outstanding)
  let outstanding = 0;
  for (const o of invoice.orders) {
    outstanding += o.amount;
  }

  recordDueDate(invoice);
  printDetails(invoice, outstanding);
}
function calculateOutstanding(invoice) {
  let outstanding = 0;
  for (const o of invoice.orders) {
    outstanding += o.amount;
  }
  return outstanding;
}
```

Como eu movi a declaração de `outstanding` para o código extraído, não preciso passá-la como parâmetro. A variável `outstanding` é a única que recebe valor novamente no código extraído, portanto posso devolvê-la.

Meu ambiente JavaScript não fornece muita informação com a compilação – na verdade, fornece menos informações do que obtenho com a análise sintática feita pelo meu editor –, portanto não há nenhum passo a ser executado nesse ponto. Minha próxima tarefa é substituir o código original por uma chamada à nova função. Como estou devolvendo o valor, preciso armazená-lo na variável original.

```
function printOwing(invoice) {
  printBanner();
  let outstanding = calculateOutstanding(invoice);
  recordDueDate(invoice);
  printDetails(invoice, outstanding);
}
function calculateOutstanding(invoice) {
  let outstanding = 0;
  for (const o of invoice.orders) {
    outstanding += o.amount;
  }
  return outstanding;
}
```

Antes de considerar que terminei, renomeio o valor de retorno para que esteja de acordo com meu estilo usual de programação.

```
function printOwing(invoice) {
  printBanner();
  const outstanding = calculateOutstanding(invoice);
  recordDueDate(invoice);
  printDetails(invoice, outstanding);
}
function calculateOutstanding(invoice) {
  let result = 0;
  for (const o of invoice.orders) {
    result += o.amount;
  }
  return result;
}
```

Também aproveito a oportunidade para modificar o outstanding *original para* const**.**

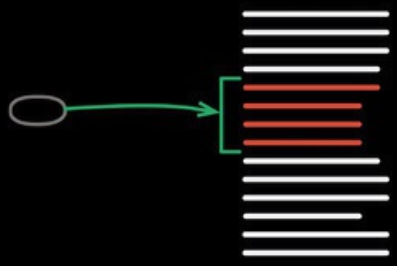
A essa altura, você pode estar se perguntando: “O que aconteceria se mais de uma variável tivesse de ser devolvida?”.

Nesse caso, tenho várias opções. Em geral, prefiro escolher um código diferente para extrair. Gosto quando uma função devolve um só valor, então eu tentaria organizar o código de modo a ter diversas funções para os diferentes valores. Se eu realmente precisar fazer a extração e houver diversos valores, posso criar um registro e devolvê-lo – em geral, porém, acho melhor reorganizar as variáveis temporárias. Nesse cenário, gosto de usar *Substituir variável temporária por consulta (Replace Temp with Query)* e *Separar variável (Split Variable)*.

Uma questão interessante surge quando estou extraindo funções que espero que sejam movidas depois para outro contexto, por exemplo, para o nível mais alto. Prefiro dar passos pequenos, portanto meu instinto diz para extrair o código em uma função aninhada antes, e depois mover essa função para o seu novo contexto. Contudo, a parte complicada é lidar com as variáveis, e essa dificuldade só será exposta quando eu mover a função. Isso mostra que, mesmo que eu extraia o código para uma função aninhada, faz sentido extraí-lo para, no mínimo, o mesmo nível da função original primeiro, a fim de que eu verifique logo se o código extraído faz sentido.

Internalizar função (Inline Function)

anteriormente: *Internalizar método* (Inline Method)
inversa de: *Extrair função (Extract Function)*



```
function getRating(driver) {
  return moreThanFiveLateDeliveries(driver) ? 2
: 1;
}

function moreThanFiveLateDeliveries(driver) {
  return driver.numberOfLateDeliveries > 5;
}
```

```
function getRating(driver) {
  return (driver.numberOfLateDeliveries > 5) ? 2
: 1;
}
```

Motivação

Um dos temas deste livro é usar funções pequenas cujos nomes mostrem o

seu propósito, pois essas funções resultam em um código mais claro e mais fácil de ler. Às vezes, porém, deparo com uma função cujo corpo é tão claro quanto o nome. Também posso refatorar o corpo do código para que fique tão claro quanto o nome. Quando isso acontece, posso me livrar da função. Um acesso indireto pode ser conveniente, mas um acesso indireto desnecessário é irritante.

Também uso *Internalizar função* quando tenho um grupo de funções que pareça estar fatorado de modo ruim. Posso internalizar todas elas em uma única função grande e, então, extrair as funções novamente do modo que eu quiser.

É comum usar *Internalizar função* quando vejo um código que utilize muito acesso indireto – quando parece que toda função faz uma simples delegação para outra função, e me perco com todas essas delegações. Talvez valha a pena ter alguns desses acessos indiretos, mas não todos. Ao fazer uma internalização, posso preservar os acessos convenientes e eliminar os demais.

Procedimento

- Verifique se esse não é um método polimórfico.

Se esse método estiver em uma classe e houver subclasses que o sobrescrevam, não poderei internalizá-lo.

- Encontre todas as chamadas da função.
- Substitua cada chamada pelo corpo da função.
- Teste após cada substituição.

A internalização completa não precisa ser feita de uma só vez. Se algumas partes forem complicadas, elas poderão ser feitas gradualmente, conforme a oportunidade permitir.

- Remova a definição da função.

Escrita dessa forma, *Internalizar função* parece simples. Em geral, ela não é. Eu poderia escrever várias páginas sobre como lidar com recursão, com vários pontos de retorno, internalizar um método em outro objeto quando não há métodos de acesso, e situações desse tipo. O motivo para eu não realizar isso é o fato de que, se essas complexidades existirem, você não deverá fazer essa refatoração.

Exemplo

No caso mais simples, essa refatoração é extremamente trivial. Começo com:

```
function rating(aDriver) {
  return moreThanFiveLateDeliveries(aDriver) ? 2 : 1;
}
function moreThanFiveLateDeliveries(aDriver) {
  return aDriver.numberOfLateDeliveries > 5;
}
```

Posso simplesmente copiar a expressão de retorno da função chamada e colá-la no ponto de chamada para substituí-la.

```
function rating(aDriver) {
  return aDriver.numberOfLateDeliveries > 5 ? 2 : 1;
}
```

Contudo, a situação pode ser um pouco mais complicada, exigindo de mim mais trabalho para adequar o código em seu novo local. Considere o caso em que começo com a pequena variação a seguir do código inicial que vimos antes:

```
function rating(aDriver) {
  return moreThanFiveLateDeliveries(aDriver) ? 2 : 1;
}
function moreThanFiveLateDeliveries(dvr) {
  return dvr.numberOfLateDeliveries > 5;
}
```

É quase igual, mas, agora, o argumento declarado em moreThanFiveLateDeliveries é diferente do nome do argumento passado. Desse modo, tenho de adaptar um pouco o código quando fizer a internalização.

```
function rating(aDriver) {
  return aDriver.numberOfLateDeliveries > 5 ? 2 : 1;
}
```

A situação pode até mesmo ser um pouco mais complicada. Considere o código a seguir:

```
function reportLines(aCustomer) {
  const lines = [];
  gatherCustomerData(lines, aCustomer);
  return lines;
}
function gatherCustomerData(out, aCustomer) {
  out.push(["name", aCustomer.name]);
  out.push(["location", aCustomer.location]);
}
```

Internalizar gatherCustomerData em reportLines não é somente uma simples operação de cortar e colar. Não é muito complicado, e, na maioria das vezes, eu faria isso em um só passo, adaptando um pouco o código. Contudo, para ser cauteloso, talvez faça sentido mover uma linha de cada vez. Assim, começo usando *Mover instruções para os pontos de chamada (Move Statements to Callers)* na primeira linha (faço isso do modo simples, cortando, colando e adaptando).

```
function reportLines(aCustomer) {
  const lines = [];
  lines.push(["name", aCustomer.name]);
  gatherCustomerData(lines, aCustomer);
  return lines;
}
function gatherCustomerData(out, aCustomer) {
  out.push(["name", aCustomer.name]);
  out.push(["location", aCustomer.location]);
}
```

Continuo então com as outras linhas, até terminar.

```
function reportLines(aCustomer) {
  const lines = [];
  lines.push(["name", aCustomer.name]);
  lines.push(["location", aCustomer.location]);
  return lines;
}
```

O ponto principal, nesse caso, é sempre estar pronto para dar passos menores. Na maioria das vezes, com as funções pequenas que geralmente escrevo, posso executar *Internalizar função* de uma só vez, ainda que haja pequenas adaptações a serem feitas no código. Porém, se deparo com complicações, trabalho com uma linha de cada vez. Mesmo com uma só linha, a situação pode se complicar um pouco; nesse caso, uso o procedimento mais sofisticado de *Mover instruções para os pontos de chamada (Move Statements to Callers)* para dividir mais ainda o trabalho. Se, me sentindo confiante, eu fizer algo do modo rápido e os testes falharem, prefiro retornar ao último estado do código em que os testes estavam passando e repito a refatoração dando passos menores, com um toque de mortificação.

Extrair variável (Extract Variable)

anteriormente: *Introduzir variável explicativa* (Introduce Explaining

Variable)
inversa de: *Internalizar variável (Inline Variable)*

```
return order.quantity * order.itemPrice -
  Math.max(0, order.quantity - 500) * order.itemPrice *
0.05 +
  Math.min(order.quantity * order.itemPrice * 0.1, 100);
```

```
const basePrice = order.quantity * order.itemPrice;
const quantityDiscount = Math.max(0, order.quantity - 500) * order.itemPrice *
0.05;
const shipping = Math.min(basePrice * 0.1, 100);
return basePrice - quantityDiscount + shipping;
```

Motivação

As expressões podem se tornar muito complexas e difíceis de ler. Em situações como essas, as variáveis locais podem ajudar a separar a expressão em partes mais fáceis de lidar. Particularmente, elas possibilitam nomear uma parte de uma lógica mais complexa. Com isso, posso entender melhor o propósito do que está acontecendo.

Variáveis como essas também são convenientes para depuração, pois oferecem um gancho fácil para um depurador usar ou um valor para uma instrução print capturar.

Se eu estiver considerando usar *Extrair variável*, significa que quero acrescentar um nome a uma expressão em meu código. Depois de decidir que quero fazer isso, também penso no contexto desse nome. Se ele for significativo somente dentro da função na qual estou trabalhando, então *Extrair variável* será uma boa opção – porém, se fizer sentido em um contexto mais amplo, considerarei deixar o nome disponível nesse contexto maior, em geral como uma função. Se o nome estiver disponível de modo mais amplo, outros códigos poderão usar essa expressão sem ter de repeti-la,

resultando em menos duplicação e melhor demonstração de meu propósito.

A desvantagem de promover o nome para um contexto mais amplo é o esforço adicional. Se esse esforço for significativamente maior, é provável que eu deixe para mais tarde, quando puder usar *Substituir variável temporária por consulta (Replace Temp with Query)*. Contudo, se for fácil, prefiro fazer a refatoração agora, para que o nome fique prontamente disponível no código. Como um bom exemplo disso, se eu estiver trabalhando com uma classe, *Extrair função (Extract Function)* será muito fácil de fazer.

Procedimento

- Garanta que a expressão que você quer extrair não tenha efeitos colaterais.
- Declare uma variável imutável. Atribua a essa variável uma cópia da expressão que você quer nomear.
- Substitua a expressão original pela nova variável.
- Teste.

Se a expressão aparecer mais de uma vez, substitua cada ocorrência pela variável, testando após cada substituição.

Exemplo

Começarei com um cálculo simples:

```
function price(order) {
  // preço é igual ao preço base – desconto por
  // quantidade (quantity discount) + frete (shipping)
  return order.quantity * order.itemPrice -
    Math.max(0, order.quantity - 500) * order.itemPrice * 0.05 +
    Math.min(order.quantity * order.itemPrice * 0.1, 100);
}
```

Apesar de ser um código simples, posso deixá-lo mais fácil ainda de entender. Em primeiro lugar, reconheço que o preço base é o produto entre a quantidade e o preço do item.

```
function price(order) {
  // preço é igual ao preço base – desconto por
  // quantidade (quantity discount) + frete (shipping)
  return order.quantity * order.itemPrice -
    Math.max(0, order.quantity - 500) * order.itemPrice * 0.05 +
    Math.min(order.quantity * order.itemPrice * 0.1, 100);
```

```
}
```

Depois que essa compreensão estiver em minha mente, coloco-a no código criando e nomeando uma variável para ela.

```
function price(order) {
  // preço é igual ao preço base – desconto por
  // quantidade (quantity discount) + frete (shipping)
  const basePrice = order.quantity * order.itemPrice;
  return order.quantity * order.itemPrice -
    Math.max(0, order.quantity - 500) * order.itemPrice * 0.05 +
    Math.min(order.quantity * order.itemPrice * 0.1, 100);
}
```

É claro que somente declarar e inicializar uma variável não tem efeito algum; também tenho de usá-la, portanto, substituo a expressão contendo o código original.

```
function price(order) {
  // preço é igual ao preço base – desconto por
  // quantidade (quantity discount) + frete (shipping)
  const basePrice = order.quantity * order.itemPrice;
  return basePrice -
    Math.max(0, order.quantity - 500) * order.itemPrice * 0.05 +
    Math.min(order.quantity * order.itemPrice * 0.1, 100);
}
```

Essa mesma expressão é usada depois, portanto posso substituí-la pela variável ali também.

```
function price(order) {
  // preço é igual ao preço base – desconto por
  // quantidade (quantity discount) + frete (shipping)
  const basePrice = order.quantity * order.itemPrice;
  return basePrice -
    Math.max(0, order.quantity - 500) * order.itemPrice * 0.05 +
    Math.min(basePrice * 0.1, 100);
}
```

A próxima linha contém o valor do desconto por quantidade (quantity discount), então posso extrair esse código também.

```
function price(order) {
  // preço é igual ao preço base – desconto por
  // quantidade (quantity discount) + frete (shipping)
  const basePrice = order.quantity * order.itemPrice;
  const quantityDiscount = Math.max(0, order.quantity - 500) * order.itemPrice * 0.05;
  return basePrice -
```

```
    quantityDiscount +
    Math.min(basePrice * 0.1, 100);
}
```

Por fim, termino com o frete (shipping). Ao fazer isso, posso remover o comentário também, pois ele não diz mais nada além do que diz o código.

```
function price(order) {
  const basePrice = order.quantity * order.itemPrice;
  const quantityDiscount = Math.max(0, order.quantity - 500) * order.itemPrice * 0.05;
  const shipping = Math.min(basePrice * 0.1, 100);
  return basePrice - quantityDiscount + shipping;
}
```

Exemplo: com uma classe

Eis o mesmo código, mas, dessa vez, no contexto de uma classe:

```
class Order {
  constructor(aRecord) {
    this._data = aRecord;
  }

  get quantity() {return this._data.quantity;}
  get itemPrice() {return this._data.itemPrice;}

  get price() {
    return this.quantity * this.itemPrice -
      Math.max(0, this.quantity - 500) * this.itemPrice * 0.05 +
      Math.min(this.quantity * this.itemPrice * 0.1, 100);
  }
}
```

Nesse caso, quero extrair os mesmos nomes, mas percebo que eles se aplicam a `Order` como um todo, e não apenas ao cálculo do preço. Como se aplicam ao pedido como um todo, estou inclinado a extrair os nomes como métodos, em vez de extraí-los como variáveis.

```
class Order {
  constructor(aRecord) {
    this._data = aRecord;
  }
  get quantity() {return this._data.quantity;}
  get itemPrice() {return this._data.itemPrice;}

  get price() {
    return this.basePrice - this.quantityDiscount + this.shipping;
  }
```

```
    get basePrice() {return this.quantity * this.itemPrice;}
    get quantityDiscount() {return Math.max(0, this.quantity - 500) * this.itemPrice * 0.05;}
    get shipping() {return Math.min(this.basePrice * 0.1, 100);}
  }
```

Essa é uma das grandes vantagens dos objetos – eles oferecem um contexto razoável para a lógica, permitindo compartilhar outras porções de lógica e de dados. Para um código tão simples como esse, não importa muito; contudo, em uma classe maior, é muito conveniente chamar porções comuns de comportamento como abstrações próprias, com os próprios nomes, para referenciá-las sempre que eu estiver trabalhando com o objeto.

Internalizar variável (Inline Variable)

anteriormente: *Internalizar variável temporária* (Inline Temp)
inversa de: *Extrair variável (Extract Variable)*

```
let basePrice =
anOrder.basePrice;
return (basePrice > 1000);
```

```
return anOrder.basePrice >
1000;
```

Motivação

As variáveis fornecem nomes para expressões em uma função e, desse modo, em geral, são boas. Às vezes, porém, o nome não diz realmente nada além do que diz a própria expressão. Em outras ocasiões, você talvez note que uma variável está atrapalhando a refatoração do código ao redor. Nesses casos, pode ser conveniente internalizar a variável.

Procedimento

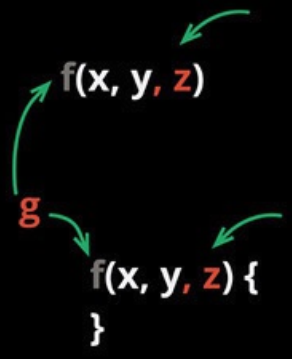
- Verifique se o lado direito da atribuição está livre de efeitos colaterais.
- Se a variável ainda não estiver declarada como imutável, faça isso e teste.

Assim, verificamos se há somente uma atribuição à variável.

- Encontre a primeira referência à variável e substitua essa ocorrência pelo lado direito da atribuição.
- Teste.
- Repita substituindo as referências à variável até ter substituído todas elas.
- Remova a declaração e a atribuição à variável.
- Teste.

Mudar declaração de função (Change Function Declaration)

também conhecida como: *Renomear função* (Rename Function)
anteriormente: *Renomear método* (Rename Method)
anteriormente: *Adicionar parâmetro* (Add Parameter)
anteriormente: *Remover parâmetro* (Remove Parameter)
também conhecida como: *Modificar assinatura* (Change Signature)




```
function circum(radius)
{...}
```

```
function circumference(radius)
{...}
```

Motivação

As funções representam o modo principal de dividir um programa em partes. As declarações de funções representam como essas partes se encaixam – elas efetivamente representam as junções em nossos sistemas de software. E, como em qualquer construção, muita coisa depende dessas junções. Boas junções me permitem acrescentar novas partes ao sistema com facilidade, mas junções ruins são uma fonte constante de problemas, tornando mais difícil entender o que o software faz e como modificá-lo quando minhas necessidades mudarem. Felizmente, o software, sendo soft, me permite modificar essas junções, desde que eu faça isso com cuidado.

O elemento mais importante de uma junção como essa é o nome da função. Um bom nome me permite entender o que a função faz quando a vejo sendo chamada, sem que eu veja o código que define a sua implementação. No entanto, pensar em bons nomes é difícil, e raramente penso nos nomes corretos na primeira tentativa. Quando vejo um nome que me deixa confuso, sinto-me tentado a deixá-lo como está – afinal de contas, é apenas um nome. Esse é um feito do maléfico demônio *Obfuscatis*; pela alma de meu programa, jamais devo dar ouvidos a ele. Se vejo uma função com um nome incorreto, é mandatório que eu o altere assim que compreender qual seria um nome melhor. Dessa forma, da próxima vez que eu ver esse código, não terei de descobrir *novamente* o que está acontecendo. (Muitas vezes, uma boa maneira de ter um nome melhor é escrever um comentário para descrever o propósito da função e, em seguida, transformar esse comentário em um nome.)

Uma lógica semelhante se aplica aos parâmetros de uma função. Os parâmetros de uma função determinam como ela se encaixa no restante de seu mundo. Os parâmetros definem o contexto no qual posso usar uma função. Se eu tiver uma função para formatar o número de telefone de uma pessoa e tal função aceitar uma pessoa como argumento, não poderei usá-la para formatar o número de telefone de uma empresa. Se eu substituir o parâmetro pessoa pelo número de telefone propriamente dito, então o código de formatação será útil de modo mais genérico.

Além de aumentar a aplicabilidade da função, posso também eliminar um pouco do acoplamento, alterando quais módulos precisam se conectar a outros. A lógica de formatação de números de telefone pode ficar em um módulo que não tenha conhecimento algum sobre pessoas. Reduzir o número

de módulos que precise conhecer um ao outro ajuda a reduzir a quantidade de informações que tenho de colocar em meu cérebro quando modifico um código – e meu cérebro não é mais tão grande quanto costumava ser (isso, porém, não diz nada sobre o tamanho de seu contêiner).

Escolher os parâmetros corretos não é algo que seja definido com regras simples. Posso ter uma função simples para determinar se um pagamento venceu, verificando se já se passaram mais de 30 dias. O parâmetro dessa função deve ser o objeto pagamento, ou a data de vencimento do pagamento? Usar o pagamento acopla a função à interface do objeto pagamento. No entanto, se uso o pagamento, posso facilmente acessar outras propriedades dele, caso a lógica evolua, sem ter de modificar todas as porções de código que chamam essa função – essencialmente, aumentando o encapsulamento da função.

A única resposta correta para esse problema é que não há uma resposta correta, especialmente com o passar do tempo. Então acho essencial conhecer *Mudar declaração de função* para que o código evolua junto com minha compreensão sobre quais devem ser as melhores junções no código.

Em geral, somente uso o nome principal de uma refatoração quando me refiro a ela em outros pontos deste livro. Entretanto, como renomear é um caso de uso muito importante para *Mudar declaração de função*, se eu estiver apenas renomeando algo, vou me referir a essa refatoração como *Renomear função* para deixar claro o que estou fazendo. Independentemente de eu estar apenas renomeando ou se estiver manipulando os parâmetros, usarei o mesmo procedimento.

Procedimento

Na maioria das refatorações deste livro, apresento somente um único procedimento. Isso não é porque há apenas um procedimento para fazer o trabalho, mas porque, em geral, um procedimento funcionará razoavelmente bem na maioria dos casos. *Mudar declaração de função*, no entanto, é uma exceção. O procedimento simples em geral é eficaz, mas há muitos casos em que uma migração mais gradual fará mais sentido. Assim, no caso dessa refatoração, olho para a mudança e me pergunto se posso modificar a declaração e todos os pontos de chamada facilmente em um só passo. Em caso afirmativo, sigo o procedimento simples. O procedimento em estilo de migração me permite modificar as chamadas de modo mais gradual – é

importante caso eu tenha muitas chamadas, caso elas sejam difíceis de alcançar, caso a função seja um método polimórfico ou se eu tiver uma modificação mais complicada na declaração.

Procedimento simples

- Se você estiver removendo um parâmetro, certifique-se de que ele não seja referenciado no corpo da função.
- Altere a declaração do método para a declaração desejada.
- Encontre todas as referências à declaração antiga do método e atualize-as com a nova chamada.
- Teste.

Em geral, é melhor separar as modificações; portanto, se quiser tanto modificar o nome quanto acrescentar um parâmetro, faça isso em passos separados. (De qualquer forma, se tiver problemas, volte e use o procedimento de migração em seu lugar.)

Procedimento de migração

- Se necessário, refatore o corpo da função para facilitar a execução do passo de extração a seguir.
- Use *Extrair função (Extract Function)* no corpo da função para criar a nova função.

Se a nova função tiver o mesmo nome que a antiga, dê um nome temporário à nova função, o qual seja fácil de procurar.

- Se a função extraída precisar de parâmetros adicionais, use o procedimento simples para adicioná-los.
- Teste.
- Aplique *Internalizar função (Inline Function)* na função antiga.
- Se você usou um nome temporário, utilize *Mudar declaração de função (Change Function Declaration)* novamente para restaurar o nome original.
- Teste.

Se você estiver alterando um método de uma classe com polimorfismo, será necessário adicionar um nível de acesso indireto a cada vinculação. Se o método for polimórfico em uma única hierarquia de classe, você só precisará do método de encaminhamento na superclasse. Se o polimorfismo não tiver

ligação na superclasse, você precisará de métodos de encaminhamento em cada classe de implementação.

Se estiver refatorando uma API publicada, poderá fazer uma pausa na refatoração depois de ter criado a nova função. Durante essa pausa, faça com que a função original se torne obsoleta e espera os clientes mudarem para a nova função. A declaração da função original poderá ser removida quando (e se) você tiver certeza de que todos os clientes da função antiga migraram para a função nova.

Exemplo: renomeando uma função (procedimento simples)

Considere a função a seguir com um nome excessivamente abreviado:

```
function circum(radius) {
  return 2 * Math.PI * radius;
}
```

Quero mudar esse nome para algo mais sensato. Começo modificando a declaração:

```
function circumference(radius) {
  return 2 * Math.PI * radius;
}
```

Em seguida, encontro todas as chamadas a `circum` e mudo o nome para `circumference`.

Diferentes ambientes de linguagens exercem impacto na facilidade para encontrar todas as referências à função antiga. Tipagem estática e um bom IDE proporcionam a melhor experiência, em geral permitindo que eu renomeie as funções automaticamente, com poucas chances de erro. Sem uma tipagem estática, isso pode ser mais complicado; até mesmo boas ferramentas de pesquisa apresentarão muitos falso-positivos.

Uso a mesma abordagem para adicionar ou remover parâmetros: localizo todas as chamadas, modifico a declaração e altero as chamadas. Em geral, é melhor fazer isso em passos separados – assim, se eu estiver renomeando a função e acrescentando um parâmetro, renomeio primeiro, testo, e então adiciono o parâmetro e testo novamente.

Uma desvantagem desse modo simples de fazer a refatoração é que preciso modificar todas as chamadas e a declaração (ou todas as declarações, em caso de polimorfismo), de uma só vez. Se houver poucas ocorrências, ou se eu

tiver ferramentas decentes de refatoração automatizadas, isso é razoável. No entanto, se houver muitas, pode ser complicado. Outro problema ocorre quando os nomes não são únicos – por exemplo, quero renomear um método changeAddress de uma classe pessoa, mas o mesmo método, que não desejo mudar, está presente em uma classe de contrato de seguro. Quanto mais complexa for a mudança, menos vontade tenho de fazê-la de uma só vez, como nesse caso. Quando esse tipo de problema surge, uso o procedimento de migração em seu lugar. De modo semelhante, se uso o procedimento simples e algo dá errado, restauro o código para o último estado conhecido em que ele estava correto e tento novamente com o procedimento de migração.

Exemplo: renomeando uma função (procedimento de migração)

Mais uma vez, tenho a função a seguir com o nome excessivamente abreviado:

```
function circum(radius) {
  return 2 * Math.PI * radius;
}
```

Para fazer essa refatoração com o procedimento de migração, começo aplicando *Extrair função (Extract Function) no corpo inteiro da função.*

```
function circum(radius) {
  return circumference(radius);
}
function circumference(radius) {
  return 2 * Math.PI * radius;
}
```

Testo esse código, e então aplico *Internalizar função (Inline Function)* na função antiga. Encontro todas as chamadas da função antiga e substituo cada uma delas por uma chamada da nova função. Posso testar após cada mudança, permitindo que eu as faça uma de cada vez. Assim que terminar, removo a função antiga.

Na maioria das refatorações, modifico um código que sou capaz de alterar, mas essa refatoração também pode ser conveniente no caso de uma API publicada – isto é, uma API usada por códigos que eu, pessoalmente, não posso mudar. Posso fazer uma pausa na refatoração depois de criar circumference e, se possível, marcar circum como obsoleta. Então espero que

quem chama a função mude e passe a usar circumference; depois que o fizerem, posso apagar circum. Mesmo que eu jamais consiga atingir a satisfatória situação de poder apagar circum, pelo menos terei um nome melhor para o novo código.

Exemplo: adicionando um parâmetro

Em um software para administrar uma biblioteca, tenho uma classe livro com capacidade para aceitar uma reserva para um usuário.

class Book…

```
addReservation(customer) {
  this._reservations.push(customer);
}
```

Devo oferecer suporte a uma fila de prioridades para as reservas. Assim, preciso de um parâmetro extra em addReservation para indicar se a reserva deve ser colocada na fila comum ou na fila de alta prioridade. Se eu puder localizar e modificar facilmente todas as chamadas, poderei simplesmente prosseguir com a mudança – mas, se não for possível, posso usar a abordagem de migração, apresentada a seguir.

Começo usando *Extrair função (Extract Function)* no corpo de addReservation para criar a nova função. Embora, em algum momento, ela vá se chamar addReservation, a função nova e a função antiga não podem coexistir com o mesmo nome. Portanto, uso um nome temporário que seja fácil de pesquisar mais tarde.

class Book…

```
addReservation(customer) {
  this.zz_addReservation(customer);
}
zz_addReservation(customer) {
  this._reservations.push(customer);
}
```

Então adiciono o parâmetro na nova declaração e em sua chamada (com efeito, usando o procedimento simples).

class Book…

```
addReservation(customer) {
  this.zz_addReservation(customer, false);
}
zz_addReservation(customer, isPriority) {
```

```
    this._reservations.push(customer);
  }
```

Quando uso JavaScript, antes de alterar qualquer uma das chamadas, gosto de aplicar *Introduzir asserção (Introduce Assertion)* para verificar se o novo parâmetro está sendo usado na chamada.

class Book…

```
  zz_addReservation(customer, isPriority) {
    assert(isPriority === true || isPriority === false);
    this._reservations.push(customer);
  }
```

Agora, quando modificar as chamadas, se eu cometer um erro e me esquecer do novo parâmetro, essa asserção me ajudará a capturar o erro. E minha longa experiência mostra que há poucos programadores mais propensos a erros do que eu.

Posso agora começar a modificar as chamadas usando *Internalizar função (Inline Function)* na função original. Isso me permite modificar uma chamada de cada vez.

Então renomeio a nova função com o nome original. Em geral, o procedimento simples funciona bem nesse caso, mas também posso usar o procedimento de migração, se necessário.

Exemplo: modificando um parâmetro para uma de suas propriedades

Os exemplos até agora eram mudanças simples de um nome e o acréscimo de um novo parâmetro, mas, com o procedimento de migração, essa refatoração é capaz de tratar casos mais complicados de forma muito organizada. A seguir, apresentarei um exemplo um pouco mais sofisticado.

Tenho uma função que determina se um cliente está na Nova Inglaterra (New England).

```
  function inNewEngland(aCustomer) {
    return ["MA", "CT", "ME", "VT", "NH", "RI"].includes(aCustomer.address.state);
  }
```

Eis uma de suas chamadas:

chamada...

```
  const newEnglanders = someCustomers.filter(c => inNewEngland(c));
```

inNewEngland usa somente o estado em que o cliente reside para determinar se ele está na Nova Inglaterra. Prefiro refatorar inNewEngland de modo que ela aceite um código de estado como parâmetro, deixando-a utilizável em outros contextos, eliminando a dependência com o cliente.

Com *Mudar declaração de função*, meu primeiro passo costumeiro é aplicar *Extrair função (Extract Function)*; nesse caso, porém, posso facilitar a tarefa refatorando um pouco o corpo da função antes. Uso *Extrair variável (Extract Variable)* no novo parâmetro que desejo utilizar.

```
function inNewEngland(aCustomer) {
  const stateCode = aCustomer.address.state;
  return ["MA", "CT", "ME", "VT", "NH", "RI"].includes(stateCode);
}
```

Agora uso *Extrair função (Extract Function)* para criar a nova função.

```
function inNewEngland(aCustomer) {
  const stateCode = aCustomer.address.state;
  return xxNEWinNewEngland(stateCode);
}
function xxNEWinNewEngland(stateCode) {
  return ["MA", "CT", "ME", "VT", "NH", "RI"].includes(stateCode);
}
```

Dou à função um nome fácil de substituir automaticamente para que ela volte a ter o nome original mais tarde. (Você pode notar que não tenho um padrão para esses nomes temporários.)

Aplico *Internalizar variável (Inline Variable)* no parâmetro de entrada na função original.

```
function inNewEngland(aCustomer) {
  return xxNEWinNewEngland(aCustomer.address.state);
}
```

Uso *Internalizar função (Inline Function)* para colocar o código da função antiga nas chamadas, efetivamente substituindo a chamada da função antiga por uma chamada da nova função. Posso fazer isso, uma chamada de cada vez.

chamada...

```
const newEnglanders = someCustomers.filter(c => xxNEWinNewEngland(c.address.state));
```

Depois de internalizar a função antiga em todas as chamadas, uso *Mudar declaração de função* novamente para alterar o nome da nova função com o

nome da função original.

chamada...

```
const newEnglanders = someCustomers.filter(c => inNewEngland(c.address.state));
```

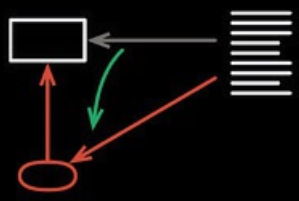
nível mais alto…

```
function inNewEngland(stateCode) {
  return ["MA", "CT", "ME", "VT", "NH", "RI"].includes(stateCode);
}
```

Ferramentas de refatoração automatizadas deixam o procedimento de migração, ao mesmo tempo, menos útil e mais eficiente. Elas o deixam menos útil porque tratam até mesmo renomeações e mudanças de parâmetros complicadas de modo mais seguro, portanto não preciso usar a abordagem de migração com tanta frequência sem esse suporte. No entanto, em casos como nesse exemplo, em que as ferramentas não conseguem fazer a refatoração completa, elas ainda facilitam bastante a operação, pois os passos principais de extrair e internalizar podem ser feitos de modo mais rápido e seguro com a ferramenta.

Encapsular variável (Encapsulate Variable)

anteriormente: *Autoencapsular campo* (Self-Encapsulate Field)
anteriormente: *Encapsular campo* (Encapsulate Field)



```
let defaultOwner = {firstName: "Martin", lastName:
"Fowler"};
```

```
let defaultOwnerData = {firstName: "Martin", lastName:
"Fowler"};
export function defaultOwner() {return defaultOwnerData;}
export function setDefaultOwner(arg) {defaultOwnerData =
arg;}
```

Motivação

A refatoração tem tudo a ver com a manipulação dos elementos de nossos programas. É mais complicado manipular dados do que manipular funções. Como usar uma função em geral significa chamá-la, posso facilmente renomear ou mover uma função enquanto mantenho a função antiga intacta como uma função de encaminhamento (desse modo, meu código antigo chama a função antiga, que chama a função nova). Em geral, não mantenho essa função de encaminhamento por muito tempo, mas ela simplifica a refatoração.

Com os dados, torna-se um pouco mais complicado porque não é possível fazer isso. Se eu mover os dados, tenho de modificar todas as referências a eles em um único ciclo para manter o código funcionando. Para dados com um escopo de acesso bem restrito, por exemplo, uma variável temporária em uma função pequena não será um problema. Contudo, à medida que o escopo aumenta, o mesmo acontece com o nível de dificuldade, e é por isso que dados globais são tão problemáticos.

Assim, se eu quiser mover dados amplamente acessados, com frequência a melhor abordagem é encapsulá-los antes, efetuando todos os acessos por meio de funções. Dessa forma, transformo a difícil tarefa de reorganizar dados na tarefa mais simples de reorganizar funções.

Encapsular dados é importante em outras situações também. O encapsulamento oferece um ponto claro para monitorar alterações e o uso dos dados; posso facilmente acrescentar validações ou uma lógica consequente nas atualizações. Tenho como hábito deixar todos os dados mutáveis encapsulados dessa forma, deixando-os acessíveis somente por meio de funções se o seu escopo for mais amplo que uma única função. Quanto maior o escopo dos dados, mais importante será encapsulá-los. Em minha abordagem para um código legado, sempre que eu tiver de alterar ou acrescentar uma nova referência a uma variável desse tipo, aproveito a oportunidade para encapsulá-la. Dessa forma, evito que haja mais acoplamento com dados comumente usados.

Esse princípio explica por que a abordagem de orientação a objetos coloca tanta ênfase em manter privados os dados de um objeto. Sempre que vejo um campo público, considero usar *Encapsular variável* (nesse caso, muitas vezes chamada de *Encapsular campo (Encapsulate Field)*) para reduzir sua visibilidade. Algumas pessoas vão além e argumentam que mesmo

referências internas a campos dentro de uma classe devem se dar por meio de funções de acesso – uma abordagem conhecida como autoencapsulamento (self-encapsulation). De modo geral, acho o autoencapsulamento um exagero – se uma classe é tão grande a ponto de ser necessário autoencapsular seus campos, ela deveria ser dividida. Contudo, autoencapsular um campo é um passo conveniente antes de dividir uma classe.

Manter dados encapsulados é muito menos importante para dados imutáveis. Quando os dados não mudam, não preciso de um lugar para colocar uma validação ou outros ganchos de lógica antes das atualizações. Também posso copiar livremente os dados em vez de movê-los – assim, não preciso alterar as referências nos locais antigos nem me preocupar se há seções de código lendo dados desatualizados. A imutabilidade é extremamente eficaz para preservar dados.

Procedimento

- Crie funções de encapsulamento para acessar e atualizar a variável.
- Execute verificações estáticas.
- Para cada referência à variável, faça uma substituição por uma chamada à função de encapsulamento apropriada. Teste após cada substituição.
- Restrinja a visibilidade da variável.

  Às vezes, não será possível impedir acesso à variável. Nesse caso, pode ser conveniente detectar quaisquer referências remanescentes renomeando a variável e testando.
- Teste.
- Se o valor da variável for um registro, considere usar *Encapsular registro (Encapsulate Record)*.

Exemplo

Considere alguns dados úteis armazenados em uma variável global.

```
let defaultOwner = {firstName: "Martin", lastName: "Fowler"};
```

Como qualquer dado, ele é referenciado por um código como este:

```
spaceship.owner = defaultOwner;
```

e é atualizado assim:

```
defaultOwner = {firstName: "Rebecca", lastName: "Parsons"};
```

Para fazer um encapsulamento básico nesses dados, começo definindo funções para ler e escrever nos dados.

```
function getDefaultOwner() {return defaultOwner;}
function setDefaultOwner(arg) {defaultOwner = arg;}
```

Então começo a trabalhar nas referências a `defaultOwner`. Quando vejo uma referência, eu a substituo por uma chamada à função de leitura.

```
spaceship.owner = getDefaultOwner();
```

Quando vejo uma atribuição, eu a substituo pela função de escrita.

```
setDefaultOwner({firstName: "Rebecca", lastName: "Parsons"});
```

Testo após cada substituição.

Quando acabar de fazer isso com todas as referências, restrinjo a visibilidade da variável. Com isso, verifico se não há outras referências que esqueci de alterar, ao mesmo tempo que garanto que futuras modificações no código não acessarão a variável diretamente. Posso fazer isso em JavaScript movendo tanto a variável quanto os métodos de acesso para o seu próprio arquivo e exportando somente os métodos de acesso.

defaultOwner.js…

```
let defaultOwner = {firstName: "Martin", lastName: "Fowler"};
export function getDefaultOwner() {return defaultOwner;}
export function setDefaultOwner(arg) {defaultOwner = arg;}
```

Se eu estiver em uma situação em que não seja possível restringir o acesso a uma variável, pode ser conveniente renomeá-la e testar novamente. Isso não impedirá acessos diretos no futuro, mas dar um nome significativo e inusitado à variável, como `__privateOnly_defaultOwner`, pode ajudar.

Não gosto de usar prefixos `get` nos getters, portanto vou renomear a função para remover esse prefixo.

defaultOwner.js…

```
let defaultOwnerData = {firstName: "Martin", lastName: "Fowler"};
export function ~~get~~defaultOwner() {return defaultOwnerData;}
export function setDefaultOwner(arg) {defaultOwnerData = arg;}
```

Uma convenção comum em JavaScript é nomear uma função de leitura e uma função de escrita do mesmo modo e diferenciá-las pela presença de um argumento. Chamo essa prática de Overloaded Getter Setter (Getter Setter Sobrecarregados) [mf-ogs], mas não gosto nem um pouco dela. Então, apesar de não gostar do prefixo `get`, manterei o prefixo `set`.

Encapsulando o valor

A refatoração básica que descrevi nesta seção encapsula uma referência a uma estrutura de dados, permitindo que eu controle seu acesso e novas atribuições. No entanto, isso não oferece controle sobre as mudanças nessa estrutura.

```
const owner1 = defaultOwner();
assert.equal("Fowler", owner1.lastName, "when set");
const owner2 = defaultOwner();
owner2.lastName = "Parsons";
assert.equal("Parsons", owner1.lastName, "after change owner2"); // está certo?
```

A refatoração básica encapsula a referência ao dado. Em muitos casos, é tudo que quero fazer naquele momento. Com frequência, porém, quero levar o encapsulamento um passo além e controlar não só as modificações na variável, mas também em seu conteúdo.

Para isso, tenho duas opções. A opção mais simples é impedir qualquer alteração no valor. Meu modo favorito de lidar com esse caso é modificar a função de leitura para que devolva uma cópia dos dados.

defaultOwner.js…

```
let defaultOwnerData = {firstName: "Martin", lastName: "Fowler"};
export function defaultOwner() {return Object.assign({}, defaultOwnerData);}
export function setDefaultOwner(arg) {defaultOwnerData = arg;}
```

Uso essa abordagem particularmente com frequência no caso de listas. Se eu devolver uma cópia dos dados, qualquer cliente que usá-la poderá modificá-la, mas essa alteração não se refletirá nos dados compartilhados. Entretanto, devo ter cuidado ao usar cópias: pode haver um código que ache que está modificando os dados compartilhados. Se for esse o caso, conto com meus testes para detectar o problema. Uma alternativa é impedir as alterações – e uma boa forma de fazer isso é usar *Encapsular registro (Encapsulate Record)*.

```
let defaultOwnerData = {firstName: "Martin", lastName: "Fowler"};
export function defaultOwner() {return new Person(defaultOwnerData);}
export function setDefaultOwner(arg) {defaultOwnerData = arg;}
class Person {
  constructor(data) {
    this._lastName = data.lastName;
    this._firstName = data.firstName
  }
  get lastName() {return this._lastName;}
```

```
get firstName() {return this._firstName;}
// e assim por diante para outras propriedades
```

A partir de agora, qualquer tentativa de atribuir novos valores às propriedades do dono default (default owner) causará um erro. Linguagens diferentes têm técnicas distintas para detectar ou impedir modificações como essa, portanto, dependendo da linguagem, eu consideraria outras opções.

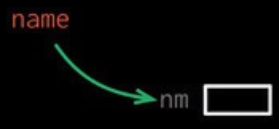
Detectar e impedir alterações como essa em geral é algo que vale a pena fazer como uma medida temporária. Posso remover as modificações ou disponibilizar funções apropriadas para elas. Então, depois que cuidar de tudo isso, posso modificar o método de leitura para que devolva uma cópia.

Até agora, falei de fazer uma cópia na leitura dos dados, mas pode valer a pena fazer uma cópia no setter também. Isso dependerá do local de onde os dados estão vindo e se é necessário manter uma ligação que reflita qualquer alteração nesses dados originais. Se eu não precisar de uma ligação como essa, uma cópia evitará acidentes como consequência de mudanças nesses dados originais. Usar uma cópia pode ser supérfluo na maioria das vezes, mas as cópias, nesse caso, geralmente têm um efeito desprezível no desempenho; por outro lado, se eu não as fizer, pode haver o risco de um período longo e difícil de depuração no futuro.

Lembre-se de que a cópia anterior e a classe wrapper só funcionam um nível abaixo na estrutura do registro. Para níveis mais profundos, é necessário ter outros níveis de cópias ou de encapsulamento do objeto.

Como você pode ver, encapsular dados é importante, embora, muitas vezes, não seja simples. O que, exatamente, deve ser encapsulado – e como fazê-lo – depende do como os dados são usados e das modificações que eu tiver em mente. Porém, quanto mais amplamente forem usados, mais valerá a pena prestar atenção para que um encapsulamento apropriado seja feito.

Renomear variável (Rename Variable)




```
let a = height *
width;
```

```
let area = height *
width;
```

Motivação

Dar bons nomes está no coração de uma programação clara. As variáveis podem fazer muito para explicar minha intenção – se eu lhes der bons nomes. No entanto, com frequência, crio nomes incorretos – às vezes porque não penso com o devido cuidado, outras porque minha compreensão sobre o problema melhora à medida que aprendo mais, e outras ainda porque o propósito do programa muda como consequência das mudanças nas necessidades de meus usuários.

Até mais que a maioria dos elementos de programa, a importância de um nome depende do quão amplamente ele é usado. Uma variável usada em uma expressão lambda de uma só linha em geral é fácil de entender – frequentemente uso uma única letra nesse caso, pois o propósito da variável será claro em seu contexto. Parâmetros de funções pequenas em geral podem ser concisos pelo mesmo motivo, embora, em uma linguagem dinamicamente tipada como JavaScript, eu goste de inserir o tipo no nome (por isso, tenho nomes de parâmetros como `aCustomer`).

Campos persistentes que perdurem além da chamada de uma única função exigem uma nomenclatura mais cuidadosa. É nesse caso que provavelmente concentro a maior parte de minha atenção.

Procedimento

- Se a variável for amplamente usada, considere usar *Encapsular variável (Encapsulate Variable)*.
- Encontre todas as referências à variável e altere cada uma delas.

Se houver referências a partir de outra base de código, a variável é uma variável publicada, e não será possível fazer essa refatoração.

Se a variável não mudar, você pode copiá-la para uma variável com um novo nome e, então, mudar gradualmente, testando após cada mudança.

- Teste.

Exemplo

O caso mais simples para renomear uma variável é quando ela é local em uma única função: é uma variável temporária ou um argumento. É trivial demais até mesmo para um exemplo: basta encontrar cada referência e alterá-la. Depois que faço isso, testo para garantir que não causei nenhuma confusão.

Problemas ocorrem quando a variável tem um escopo mais amplo do que uma única função. Pode haver várias referências espalhadas por toda a base de código:

```
let tpHd = "untitled";
```

Algumas referências acessam a variável:

```
result += `<h1>${tpHd}</h1>`;
```

Outras a atualizam:

```
tpHd = obj['articleTitle'];
```

Minha resposta costumeira para essa situação é aplicar *Encapsular variável (Encapsulate Variable)*.

```
result += `<h1>${title()}</h1>`;
setTitle(obj['articleTitle']);
function title() {return tpHd;}
function setTitle(arg) {tpHd = arg;}
```

Nesse ponto, posso renomear a variável.

```
let _title = "untitled";
function title() {return _title;}
function setTitle(arg) {_title = arg;}
```

Eu poderia prosseguir internalizando as funções de encapsulamento, de modo que todas as chamadas usem a variável diretamente. Raramente, porém, faço isso. Se a variável for usada de forma suficientemente ampla, a ponto de eu sentir necessidade de encapsulá-la para modificar o seu nome, vale a pena mantê-la encapsulada por funções com vistas ao futuro.

Nos casos em que faria a internalização, eu chamaria a função de leitura de `getTitle` e não usaria um underscore no nome de variável ao renomeá-la.

Renomeando uma constante

Se eu estiver renomeando uma constante (ou algo que atue como uma constante para os clientes), posso evitar um encapsulamento, e mesmo assim

renomear gradualmente, por meio de cópias. Se a declaração original tiver o seguinte aspecto:

```
const cpyNm = "Acme Gooseberries";
```

Posso iniciar o processo de renomear criando uma cópia:

```
const companyName = "Acme Gooseberries";
const cpyNm = companyName;
```

Com a cópia, posso gradualmente modificar as referências, do nome antigo para o nome novo. Quando terminar, removo a cópia. Prefiro declarar o novo nome e copiá-lo para o nome antigo se isso facilitar um pouco a remoção do nome antigo e restaurá-lo caso um teste falhe.

Isso funciona para constantes bem como para variáveis que sejam somente de leitura para os clientes (por exemplo, uma variável exportada em JavaScript).

Introduzir objeto de parâmetros (Introduce Parameter Object)


```
function amountInvoiced(startDate, endDate)
{...}
function amountReceived(startDate, endDate)
{...}
function amountOverdue(startDate, endDate)
{...}
```

```
function amountInvoiced(aDateRange)
{...}
function amountReceived(aDateRange)
{...}
function amountOverdue(aDateRange)
{...}
```

Motivação

Muitas vezes, vejo grupos de dados que estão regularmente juntos, aparecendo em uma função após outra. Um grupo como esse forma um agrupamento de dados, e gosto de substituí-lo por uma única estrutura de dados.

Agrupar dados em uma estrutura é importante porque torna explícito o relacionamento entre eles. Com isso, reduzo o tamanho das listas de parâmetros para qualquer função que use a nova estrutura. A estrutura contribui para que haja consistência, pois todas as funções que a utilizem terão os mesmos nomes para acessar seus elementos.

No entanto, a verdadeira eficácia dessa refatoração está no modo como ela permite modificações mais profundas no código. Quando identifico essas novas estruturas, posso redirecionar o comportamento do programa a fim de usá-las. Crio funções que captam o comportamento comum nesses dados – seja como um conjunto de funções comuns ou como uma classe que combine a estrutura de dados com essas funções. Esse processo pode alterar o quadro conceitual do código, elevando essas estruturas a novas abstrações que podem simplificar bastante a minha compreensão sobre o domínio. Quando isso funciona, os efeitos podem ser surpreendentemente eficazes – mas nada disso será possível, a menos que eu use *Introduzir objeto de parâmetros* para iniciar o processo.

Procedimento

- Se ainda não houver uma estrutura apropriada, crie uma.

  Prefiro usar uma classe, pois isso facilita agrupar comportamentos mais tarde. Em geral, gosto de garantir que essas estruturas sejam objetos de valor (value objects) [mf-vo].
- Teste.
- Use *Mudar declaração de função (Change Function Declaration)* para acrescentar um parâmetro para a nova estrutura.
- Teste.
- Adapte cada chamada para que passe a instância correta da nova estrutura. Teste após cada adaptação.
- Para cada elemento da nova estrutura, substitua o uso do parâmetro original pelo elemento da estrutura. Remova o parâmetro. Teste.

Exemplo

Começarei com um código que analisa um conjunto de temperatura lidas e determina se alguma delas está fora de um intervalo de operação. Eis a aparência dos dados lidos:

```
const station = { name: "ZB1",
                  readings: [
                    {temp: 47, time: "2016-11-10 09:10"},
                    {temp: 53, time: "2016-11-10 09:20"},
                    {temp: 58, time: "2016-11-10 09:30"},
                    {temp: 53, time: "2016-11-10 09:40"},
                    {temp: 51, time: "2016-11-10 09:50"},
                  ]
                };
```

Tenho uma função para encontrar as leituras que estejam fora de um intervalo de temperaturas.

```
function readingsOutsideRange(station, min, max) {
  return station.readings
    .filter(r => r.temp < min || r.temp > max);
}
```

Ela pode ser chamada por um código como este:

chamada

```
alerts = readingsOutsideRange(station,
                              operatingPlan.temperatureFloor,
                              operatingPlan.temperatureCeiling);
```

Observe como o código da chamada obtém os dois dados como um par a partir de outro objeto e passa o par para readingsOutsideRange. O plano de operação (operating plan) usa nomes diferentes para representar o início e o fim do intervalo se comparado com readingsOutsideRange. Um intervalo como esse é um caso comum em que seria melhor se os dois dados separados fossem combinados em um único objeto. Começarei declarando uma classe para os dados combinados.

```
class NumberRange {
  constructor(min, max) {
    this._data = {min: min, max: max};
  }
  get min() {return this._data.min;}
  get max() {return this._data.max;}
}
```

Declaro uma classe em vez de simplesmente usar um objeto JavaScript básico, pois, em geral, percebo que essa refatoração é o primeiro passo para mover comportamentos para o objeto recém-criado. Como uma classe faz sentido nesse caso, uso uma de imediato. Também não disponibilizo nenhum método de atualização na nova classe, pois é provável que eu faça dela um Objeto de Valor (Value Object) [mf-vo]. Na maioria das vezes em que faço essa refatoração, crio objetos de valor.

Então uso *Mudar declaração de função (Change Function Declaration)* para acrescentar o novo objeto como um parâmetro de readingsOutsideRange.

```
function readingsOutsideRange(station, min, max, range) {
  return station.readings
    .filter(r => r.temp < min || r.temp > max);
}
```

Em JavaScript, posso deixar a chamada como está, mas, em outras linguagens, eu teria de acrescentar um novo parâmetro nulo; seria algo como:

chamada

```
alerts = readingsOutsideRange(station,
                              operatingPlan.temperatureFloor,
                              operatingPlan.temperatureCeiling,
                              null);
```

Até agora, não alterei nenhum comportamento e os testes devem continuar passando. Então acesso cada chamada e faço uma adequação para que o intervalo correto de dados seja passado.

chamada

```
const range = new NumberRange(operatingPlan.temperatureFloor,
operatingPlan.temperatureCeiling);
alerts = readingsOutsideRange(station,
                              operatingPlan.temperatureFloor,
                              operatingPlan.temperatureCeiling,
                              range);
```

Ainda não alterei nenhum comportamento, pois o parâmetro não está sendo usado. Todos os testes devem continuar passando.

Agora posso começar a fazer substituições no uso dos parâmetros. Começarei com o máximo.

```
function readingsOutsideRange(station, min, ~~max,~~ range) {
  return station.readings
    .filter(r => r.temp < min || r.temp > range.max);
}
```

chamada

```
const range = new NumberRange(operatingPlan.temperatureFloor,
operatingPlan.temperatureCeiling);
alerts = readingsOutsideRange(station,
                              operatingPlan.temperatureFloor,
                              operatingPlan.temperatureCeiling,
                              range);
```

Posso testar nesse ponto, e então removo o outro parâmetro.

```
function readingsOutsideRange(station, min, range) {
  return station.readings
    .filter(r => r.temp < range.min || r.temp > range.max);
}
```

chamada

```
const range = new NumberRange(operatingPlan.temperatureFloor,
operatingPlan.temperatureCeiling);
alerts = readingsOutsideRange(station,
                              operatingPlan.temperatureFloor,
                              range);
```

Com isso, concluo essa refatoração. No entanto, substituir um grupo de parâmetros por um objeto real é apenas uma preparação para a parte realmente boa. A grande vantagem de criar uma classe como essa é que posso mover comportamentos para a nova classe. Neste exemplo, eu acrescentaria um método que teste se um valor está dentro do intervalo.

```
function readingsOutsideRange(station, range) {
  return station.readings
    .filter(r => !range.contains(r.temp));
}
```

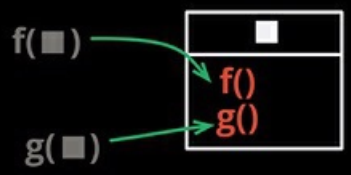
class NumberRange…

```
  contains(arg) {return (arg >= this.min && arg <= this.max);}
```

Esse é um primeiro passo para criar um intervalo [mf-range] que poderá assumir muitos comportamentos úteis. Depois de ter identificado a necessidade de um intervalo em meu código, posso ficar constantemente procurando outros casos em que há um par de números máx./mín. e substituí-los por um intervalo. (Uma possibilidade imediata é o plano de operação, substituindo temperatureFloor e temperatureCeiling por um temperatureRange.) À medida que observo como esses pares são usados, posso mover outros comportamentos úteis para a classe de intervalo, simplificando seu uso em toda a base de código. Um dos primeiros recursos que posso adicionar é um

método de igualdade baseado em valores para transformá-lo em um verdadeiro objeto de valor.

Combinar funções em classe (Combine Functions into Class)




```
function base(aReading) {...}
function taxableCharge(aReading) {...}
function calculateBaseCharge(aReading)
{...}
```

```
class Reading {
  base() {...}
  taxableCharge() {...}
  calculateBaseCharge()
{...}
}
```

Motivação

As classes são uma construção fundamental na maioria das linguagens de programação modernas. Elas associam dados e funções em um ambiente compartilhado, expondo alguns desses dados e funções a outros elementos do programa a fim de que haja colaboração. São a construção principal nas linguagens orientadas a objetos, mas são também úteis em outras abordagens.

Quando vejo um grupo de funções que atua de forma muito próxima em um corpo comum de dados (em geral, passados como argumentos da chamada de função), percebo uma oportunidade para criar uma classe. Usar uma classe deixa o ambiente comum compartilhado por essas funções mais explícito, permite simplificar as chamadas de função dentro do objeto por meio da remoção de vários argumentos, além de fornecer uma referência para a passagem de um objeto como esse a outras partes do sistema.

Além de organizar funções já criadas, essa refatoração também representa uma boa oportunidade para identificar outras porções de processamento e refatorá-las em métodos na nova classe.

Outra maneira de organizar funções juntas é usar *Combinar funções em transformação (Combine Functions into Transform)*. Qual delas deve ser usada depende do contexto mais amplo do programa. Uma vantagem significativa de usar uma classe é o fato de ela permitir aos clientes alterar os dados nucleares do objeto, e as derivações permanecem consistentes.

Assim como em uma classe, funções como essas também podem ser combinadas em uma função aninhada. Em geral, prefiro uma classe a uma função aninhada, pois pode ser difícil testar funções aninhadas. As classes também são necessárias quando há mais de uma função no grupo que eu queira expor aos colaboradores.

Linguagens que não tenham classes como um elemento de primeira classe, mas têm funções de primeira classe, em geral usam Função como Objeto (Function As Object) [mf-fao] para oferecer esse recurso.

Procedimento

- Aplique *Encapsular registro (Encapsulate Record)* no registro de dados comuns compartilhado pelas funções.

Se os dados que forem comuns entre as funções ainda não estiverem agrupados em uma estrutura de registro, use *Introduzir objeto de parâmetros (Introduce Parameter Object)* para criar um registro e agrupá-los.

- Em cada função que use o registro comum, utilize *Mover função (Move Function)* para movê-la para a nova classe.

Qualquer argumento da chamada da função que seja membro da classe poderá ser removido da lista de argumentos.

- Cada parte da lógica que manipule os dados pode ser extraída com *Extrair função (Extract Function)* e, em seguida, pode ser movida para a nova classe.

Exemplo

Eu cresci na Inglaterra, um país famoso por seu amor pelo chá. (Pessoalmente não gosto da maioria dos chás servidos na Inglaterra, mas

adquiri gosto pelos chás chineses e japoneses.) Então, minha fantasia de autor concebeu um serviço público para fornecer chá para a população. Todo mês, os medidores de chá são lidos a fim de obter um registro como este:

```
reading = {customer: "ivan", quantity: 10, month: 5, year: 2017};
```

Observo o código que processa esses registros e vejo muitos lugares em que cálculos semelhantes são feitos com os dados. Desse modo, encontro um local que calcula o preço básico cobrado (base charge):

cliente 1…

```
const aReading = acquireReading();
const baseCharge = baseRate(aReading.month, aReading.year) * aReading.quantity;
```

Sendo a Inglaterra, tudo que é essencial deve estar sujeito à cobrança de impostos, portanto isso vale para o chá. Contudo, as leis permitem uma quantidade essencial mínima de chá livre de impostos.

cliente 2…

```
const aReading = acquireReading();
const base = (baseRate(aReading.month, aReading.year) * aReading.quantity);
const taxableCharge = Math.max(0, base - taxThreshold(aReading.year));
```

Tenho certeza de que, assim como eu, você deve ter percebido que a fórmula para o preço básico cobrado está duplicada nesses dois fragmentos. Se você for como eu, já estará prestes a lançar mão de *Extrair função (Extract Function)*. O interessante é que nosso trabalho foi feito para nós em outro lugar.

cliente 3…

```
const aReading = acquireReading();
const basicChargeAmount = calculateBaseCharge(aReading);
function calculateBaseCharge(aReading) {
  return baseRate(aReading.month, aReading.year) * aReading.quantity;
}
```

Considerando isso, tenho um impulso natural de alterar as duas porções anteriores de código de cliente a fim de usar essa função. Contudo, o problema com funções de nível mais alto como essa é que se torna fácil nos esquecermos delas. Prefiro modificar o código para fazer com que a função tenha uma conexão mais próxima com os dados que ela processa. Uma boa maneira de fazer isso é transformar os dados em uma classe.

Para transformar o registro em uma classe, uso *Encapsular registro (Encapsulate Record)*.

```
class Reading {
  constructor(data) {
    this._customer = data.customer;
    this._quantity = data.quantity;
    this._month = data.month;
    this._year = data.year;
  }
  get customer() {return this._customer;}
  get quantity() {return this._quantity;}
  get month() {return this._month;}
  get year() {return this._year;}
}
```

Para transferir o comportamento, começarei com a função que já tenho: calculateBaseCharge. Para usar a nova classe, preciso aplicá-la aos dados assim que os adquirir.

cliente 3…

```
const rawReading = acquireReading();
const aReading = new Reading(rawReading);
const basicChargeAmount = calculateBaseCharge(aReading);
```

Então uso *Mover função (Move Function)* para mover calculateBaseCharge para a nova classe.

class Reading…

```
get calculateBaseCharge() {
  return baseRate(this.month, this.year) * this.quantity;
}
```

cliente 3…

```
const rawReading = acquireReading();
const aReading = new Reading(rawReading);
const basicChargeAmount = aReading.calculateBaseCharge;
```

Enquanto faço isso, utilizo *Renomear função (Rename Function)* para deixá-la mais de acordo com minha preferência.

```
get baseCharge() {
  return baseRate(this.month, this.year) * this.quantity;
}
```

cliente 3…

```
const rawReading = acquireReading();
const aReading = new Reading(rawReading);
const basicChargeAmount = aReading.baseCharge;
```

Com esse nome, o cliente da classe de leitura não poderá dizer se o preço básico cobrado é um campo ou um valor derivado. Isso é bom – é o Princípio do Acesso Uniforme (Uniform Access Principle) [mf-ua].

Agora altero o primeiro cliente para que chame o método em vez de repetir o cálculo.

cliente 1…

```
const rawReading = acquireReading();
const aReading = new Reading(rawReading);
const baseCharge = aReading.baseCharge;
```

Há uma boa chance de que eu use *Internalizar variável (Inline Variable)* na variável `baseCharge` antes de terminar o serviço. Mais relevante para essa refatoração, porém, é o cliente que calcula o valor sujeito a impostos. Meu primeiro passo nesse caso é usar a nova propriedade com o preço básico cobrado.

cliente 2…

```
const rawReading = acquireReading();
const aReading = new Reading(rawReading);
const taxableCharge = Math.max(0, aReading.baseCharge - taxThreshold(aReading.year));
```

Uso *Extrair função (Extract Function)* no cálculo do valor sujeito a impostos.

```
function taxableChargeFn(aReading) {
  return Math.max(0, aReading.baseCharge - taxThreshold(aReading.year));
}
```

cliente 3…

```
const rawReading = acquireReading();
const aReading = new Reading(rawReading);
const taxableCharge = taxableChargeFn(aReading);
```

Então aplico *Mover função (Move Function)*.

class Reading…

```
get taxableCharge() {
  return Math.max(0, this.baseCharge - taxThreshold(this.year));
}
```

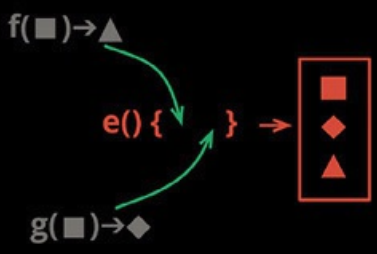
cliente 3…

```
const rawReading = acquireReading();
const aReading = new Reading(rawReading);
const taxableCharge = aReading.taxableCharge;
```

Como todos os dados derivados são calculados por demanda, não terei

problemas caso tenha de atualizar os dados armazenados. Em geral, prefiro dados imutáveis, mas várias circunstâncias nos forçam a trabalhar com dados mutáveis (por exemplo, em JavaScript, um ecossistema de linguagem que não foi projetado com a imutabilidade em mente). Se houver uma chance razoável de os dados serem atualizados em algum ponto do programa, uma classe será bastante conveniente.

Combinar funções em transformação (Combine Functions into Transform)




```
function base(aReading) {...}
function taxableCharge(aReading)
{...}
```

```
function enrichReading(argReading) {
  const aReading = _.cloneDeep(argReading);
  aReading.baseCharge = base(aReading);
  aReading.taxableCharge =
taxableCharge(aReading);
  return aReading;
}
```

Motivação

Um software muitas vezes envolve alimentar dados para programas que calculam várias informações derivadas a partir deles. Esses valores derivados podem ser necessários em diversos lugares, e esses cálculos com frequência se repetem nos pontos em que os dados derivados são usados. Prefiro reunir todos esses dados derivados de modo a ter um local consistente para encontrá-los e atualizá-los, evitando qualquer lógica duplicada.

Uma forma de fazer isso é usar uma função de dados que aceite os dados originais como entrada e calcule todos os dados derivados, colocando cada um desses valores em um campo nos dados de saída. Então, para analisar os dados derivados, tudo que preciso fazer é olhar para a função de transformação.

Uma alternativa para *Combinar funções em transformação* é usar *Combinar funções em classe (Combine Functions into Class)*, a qual move a lógica para os métodos de uma classe criada a partir dos dados originais. Qualquer uma dessas refatorações é útil, e minha opção muitas vezes dependerá do estilo de programação que já estiver no software. Contudo, há uma diferença importante: usar uma classe é muito melhor se os dados originais forem atualizados no código. Usar uma transformação armazena dados derivados no novo registro, portanto, se os dados originais mudarem, haverá inconsistências.

Um dos motivos pelos quais gosto de combinar funções é evitar duplicação na lógica de derivação. Posso fazer isso apenas usando *Extrair função (Extract Function)* na lógica; em geral, porém, é difícil encontrar as funções, a menos que elas sejam mantidas próximas das estruturas de dados nas quais atuam. Usar uma transformação (ou uma classe) facilita encontrá-las e usá-las.

Procedimento

- Crie uma função de transformação que aceite o registro a ser transformado e devolva os mesmos valores.

Em geral, isso envolverá uma cópia profunda (deep copy) do registro. Muitas vezes, vale a pena escrever um teste para garantir que a transformação não altera o registro original.

- Escolha uma lógica e mova seu corpo para a transformação a fim de criar um novo campo no registro. Altere o código do cliente para que acesse o novo campo.

Se a lógica for complexa, use *Extrair função (Extract Function)* antes.

- Teste.
- Repita para as demais funções relevantes.

Exemplo

No lugar em que cresci, o chá é uma parte importante da vida – tanto que posso imaginar um serviço especial que forneça chá para a população, administrado como se fosse um serviço público. Todo mês, o serviço faz uma leitura da quantidade de chá adquirida por um consumidor.

```
reading = {customer: "ivan", quantity: 10, month: 5, year: 2017};
```

O código em vários lugares calcula as diversas consequências desse uso do chá. Um desses cálculos é o custo básico usado para calcular o valor cobrado do consumidor.

cliente 1…

```
const aReading = acquireReading();
const baseCharge = baseRate(aReading.month, aReading.year) * aReading.quantity;
```

Outro cálculo é o valor sujeito à cobrança de impostos – é menor que o valor básico, pois o governo sabiamente considera que todo cidadão pode ter um pouco de chá livre de impostos.

cliente 2…

```
const aReading = acquireReading();
const base = (baseRate(aReading.month, aReading.year) * aReading.quantity);
const taxableCharge = Math.max(0, base - taxThreshold(aReading.year));
```

Observando esse código, vejo esses cálculos se repetirem em vários lugares. Duplicações como essas estão pedindo por problemas quando tiverem de ser alteradas (e aposto no “quando”, e não no “se”). Posso lidar com essa repetição usando *Extrair função (Extract Function)* nesses cálculos, mas funções como essas em geral acabam espalhadas por todo o programa, fazendo com que seja difícil para os futuros desenvolvedores perceberem que elas existem. De fato, olhando ao redor, descubro que há uma função desse tipo, utilizada em outra parte do código.

cliente 3…

```
const aReading = acquireReading();
const basicChargeAmount = calculateBaseCharge(aReading);
function calculateBaseCharge(aReading) {
  return baseRate(aReading.month, aReading.year) * aReading.quantity;
}
```

Uma forma de lidar com isso é mover todas essas derivações para um passo de transformação que tome as leituras brutas e gere uma leitura enriquecida com todos os resultados derivados comuns.

Começo criando uma função de transformação que simplesmente copia o

objeto de entrada.

```
function enrichReading(original) {
  const result = _.cloneDeep(original);
  return result;
}
```

Estou usando o cloneDeep *da biblioteca* lodash *para criar uma cópia profunda.*

Quando aplico uma transformação que gera essencialmente os mesmos dados, mas com informações adicionais, gosto de nomeá-la usando "enrich" (enriquecer). Se ela gerasse algo que considero ser diferente, eu a nomearia com "transform" (transformar).

Em seguida, escolho um dos cálculos que quero alterar. Em primeiro lugar, enriqueço a leitura usada com a atual, que ainda não faz nada.

cliente 3…

```
const rawReading = acquireReading();
const aReading = enrichReading(rawReading);
const basicChargeAmount = calculateBaseCharge(aReading);
```

Uso *Mover função (Move Function)* em calculateBaseCharge para movê-la para o cálculo com enriquecimento.

```
function enrichReading(original) {
  const result = _.cloneDeep(original);
  result.baseCharge = calculateBaseCharge(result);
  return result;
}
```

Na função de transformação, fico satisfeito em modificar um objeto de resultado, em vez de copiá-lo a cada vez. Gosto da imutabilidade, mas a maioria das linguagens comuns dificulta o seu uso. Estou disposto a fazer o esforço extra para aceitá-la nas fronteiras, mas aceito mutações em escopos menores. Também escolho meus nomes (usando aReading como a variável acumuladora) para facilitar mover o código para a função de transformação.

Modifico o cliente que usa essa função de modo a utilizar o campo enriquecido.

cliente 3…

```
const rawReading = acquireReading();
const aReading = enrichReading(rawReading);
const basicChargeAmount = aReading.baseCharge;
```

Depois de ter movido todas as chamadas para calculateBaseCharge, posso aninhá-la em enrichReading. Isso deixaria claro que os clientes que precisem calcular o

custo básico devem usar o registro enriquecido.

Há uma armadilha da qual você deve estar ciente nesse caso. Quando escrevo enrichReading desse modo, para devolver a leitura enriquecida, estou supondo que o registro de leitura original não está sendo alterado. Portanto, adicionar um teste é uma atitude que considero sábia.

```
it('check reading unchanged', function() {
  const baseReading = {customer: "ivan", quantity: 15, month: 5, year: 2017};
  const oracle = _.cloneDeep(baseReading);
  enrichReading(baseReading);
  assert.deepEqual(baseReading, oracle);
});
```

Posso, então, modificar o cliente 1 para que use também o mesmo campo.

cliente 1…

```
const rawReading = acquireReading();
const aReading = enrichReading(rawReading);
const baseCharge = aReading.baseCharge;
```

Há uma boa chance de que eu possa então usar *Internalizar variável (Inline Variable)* em baseCharge também.

Agora concentro minha atenção no cálculo do valor sujeito a impostos. Meu primeiro passo é acrescentar a função de transformação.

```
const rawReading = acquireReading();
const aReading = enrichReading(rawReading);
const base = (baseRate(aReading.month, aReading.year) * aReading.quantity);
const taxableCharge = Math.max(0, base - taxThreshold(aReading.year));
```

Posso substituir imediatamente o cálculo do custo básico pelo novo campo. Se o cálculo fosse complexo, eu poderia usar *Extrair função (Extract Function)* antes, mas, nesse caso, é simples o bastante para que tudo seja feito em um só passo.

```
const rawReading = acquireReading();
const aReading = enrichReading(rawReading);
const base = aReading.baseCharge;
const taxableCharge = Math.max(0, base - taxThreshold(aReading.year));
```

Depois que eu testar e constatar que esse código funciona, aplico *Internalizar variável (Inline Variable)*:

```
const rawReading = acquireReading();
const aReading = enrichReading(rawReading);
const taxableCharge = Math.max(0, aReading.baseCharge - taxThreshold(aReading.year));
```

e passo esse cálculo para a função de transformação:

```
function enrichReading(original) {
  const result = _.cloneDeep(original);
  result.baseCharge = calculateBaseCharge(result);
  result.taxableCharge = Math.max(0, result.baseCharge - taxThreshold(result.year));
  return result;
}
```

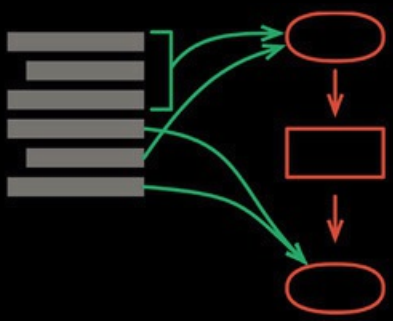
Modifico o código original para que use o novo campo.

```
const rawReading = acquireReading();
const aReading = enrichReading(rawReading);
const taxableCharge = aReading.taxableCharge;
```

Depois de testar esse código, é provável que eu possa usar *Internalizar variável (Inline Variable)* em taxableCharge.

Um grande problema com uma leitura enriquecida como essa é: o que aconteceria se um cliente modificasse um dado? Modificar, por exemplo, o campo de quantidade resultaria em dados inconsistentes. Para evitar isso em JavaScript, minha melhor opção é usar *Combinar funções em classe (Combine Functions into Class)*. Se eu estiver usando uma linguagem com estruturas de dados imutáveis, não terei esse problema; assim, é mais comum ver transformações nessas linguagens. Contudo, mesmo em linguagens sem imutabilidade, posso usar transformações caso os dados apareçam em um contexto somente de leitura, por exemplo, com dados derivados para exibição em uma página web.

Separar em fases (Split Phase)



```
const orderData = orderString.split(/\s+/);
const productPrice = priceList[orderData[0].split("-")
[1]];
const orderPrice = parseInt(orderData[1]) *
```

```
productPrice;
```

```
const orderRecord = parseOrder(order);
const orderPrice = price(orderRecord, priceList);

function parseOrder(aString) {
  const values = aString.split(/\s+/);
  return ({
    productID: values[0].split("-")[1],
    quantity: parseInt(values[1]),
  });
}
function price(order, priceList) {
  return order.quantity *
priceList[order.productID];
}
```

Motivação

Quando deparo com um código que lida com duas tarefas diferentes, procuro uma forma de separá-las em módulos distintos. Eu me empenho em fazer essa separação porque, caso precise fazer uma alteração, poderei lidar com cada assunto separadamente, sem ter de manter ambos em minha mente ao mesmo tempo. Se estiver com sorte, talvez eu tenha de modificar apenas um módulo, sem precisar me lembrar de nenhum dos detalhes do outro módulo.

Uma das formas mais organizadas de fazer uma separação como essa é dividir o comportamento em duas fases sequenciais. Um bom exemplo disso é quando há um processamento cujas entradas não refletem o modelo necessário para executar a lógica. Antes de começar, você pode transformar os dados de entrada em um formato conveniente para o seu processamento principal. Ou pode tomar a lógica que deve ser executada e separá-la em passos sequenciais, em que cada passo é significativamente diferente quanto ao que é feito.

O exemplo mais óbvio disso é um compilador. Sua tarefa básica é tomar um texto (código em uma linguagem de programação) e transformá-lo em um formato executável (por exemplo, um código-objeto para um hardware específico). Com o tempo, percebemos que isso pode ser convenientemente separado em uma cadeia de fases: transformar o texto em tokens, fazer parse

dos tokens para gerar uma árvore sintática, executar então vários passos para transformar a árvore sintática (por exemplo, para otimização) e, por fim, gerar o código-objeto. Cada passo tem um escopo limitado, e posso pensar em um passo sem compreender os detalhes dos outros passos.

Separar em fases dessa forma é comum em um software de grande porte; cada uma das várias fases de um compilador pode conter muitas funções e classes. No entanto, posso executar a refatoração básica de separação em fases em qualquer fragmento de código – sempre que vir uma oportunidade para separar o código em diferentes fases de forma conveniente. A melhor pista é quando diferentes etapas do fragmento de código usam conjuntos de dados e funções distintos. Ao transformá-los em módulos separados, posso deixar essa diferença explícita, revelando a diferença no código.

Procedimento

- Extraia o código da segunda fase em sua própria função.
- Teste.
- Introduza uma estrutura de dados intermediária como um argumento adicional para a função extraída.
- Teste.
- Analise cada parâmetro da segunda fase extraída. Se ele for usado pela primeira fase, mova-o para a estrutura de dados intermediária. Teste após cada alteração.

Às vezes, um parâmetro não deve ser usado pela segunda fase. Nesse caso, extraia o resultado de cada uso do parâmetro para um campo da estrutura de dados intermediária e use *Mover instruções para os pontos de chamada (Move Statements to Callers)* na linha que o preenche.

- Aplique *Extrair função (Extract Function)* no código da primeira fase, devolvendo a estrutura de dados intermediária.

Também é razoável extrair a primeira fase em um objeto de transformação.

Exemplo

Começarei com um código para atribuir preço a um pedido referente a um tipo vago e insignificante de produto:

```
function priceOrder(product, quantity, shippingMethod) {
  const basePrice = product.basePrice * quantity;
```

```
  const discount = Math.max(quantity - product.discountThreshold, 0)
        * product.basePrice * product.discountRate;
  const shippingPerCase = (basePrice > shippingMethod.discountThreshold)
        ? shippingMethod.discountedFee : shippingMethod.feePerCase;
  const shippingCost = quantity * shippingPerCase;
  const price = basePrice - discount + shippingCost;
  return price;
}
```

Embora esse seja o tipo comum de exemplo trivial, há uma ideia de duas fases nesse cenário. As duas primeiras linhas de código usam informações do produto para calcular o preço dependente do produto no pedido, enquanto o código seguinte utiliza informações de frete para determinar o seu custo. Se eu tiver alterações futuras que compliquem os cálculos dos preços e do frete, mas que funcionem de modo relativamente independente, então separar esse código em duas fases será importante.

Começo aplicando *Extrair função (Extract Function)* no cálculo do frete.

```
function priceOrder(product, quantity, shippingMethod) {
  const basePrice = product.basePrice * quantity;
  const discount = Math.max(quantity - product.discountThreshold, 0)
        * product.basePrice * product.discountRate;
  const price = applyShipping(basePrice, shippingMethod, quantity, discount);
  return price;
}
function applyShipping(basePrice, shippingMethod, quantity, discount) {
  const shippingPerCase = (basePrice > shippingMethod.discountThreshold)
        ? shippingMethod.discountedFee : shippingMethod.feePerCase;
  const shippingCost = quantity * shippingPerCase;
  const price = basePrice - discount + shippingCost;
  return price;
}
```

Passo todos os dados necessários a essa segunda fase como parâmetros individuais. Em um caso mais realista, pode haver muitos deles, mas não me preocupo com isso, pois me livrarei deles depois.

A seguir, introduzo a estrutura de dados intermediária que fará a comunicação entre as duas fases.

```
function priceOrder(product, quantity, shippingMethod) {
  const basePrice = product.basePrice * quantity;
  const discount = Math.max(quantity - product.discountThreshold, 0)
        * product.basePrice * product.discountRate;
  const priceData = {};
```

```
  const price = applyShipping(priceData, basePrice, shippingMethod, quantity, discount);
  return price;
}

function applyShipping(priceData, basePrice, shippingMethod, quantity, discount) {
  const shippingPerCase = (basePrice > shippingMethod.discountThreshold)
        ? shippingMethod.discountedFee : shippingMethod.feePerCase;
  const shippingCost = quantity * shippingPerCase;
  const price = basePrice - discount + shippingCost;
  return price;
}
```

Observo agora os vários parâmetros para applyShipping. O primeiro é basePrice, criado pelo código da primeira fase. Então, passo-o para a estrutura de dados intermediária, removendo-o da lista de parâmetros.

```
function priceOrder(product, quantity, shippingMethod) {
  const basePrice = product.basePrice * quantity;
  const discount = Math.max(quantity – product.discountThreshold, 0)
        * product.basePrice * product.discountRate;
  const priceData = {basePrice: basePrice};
  const price = applyShipping(priceData, ~~basePrice,~~ shippingMethod, quantity, discount);
  return price;
}
function applyShipping(priceData, ~~basePrice,~~ shippingMethod, quantity, discount) {
  const shippingPerCase = (priceData.basePrice > shippingMethod.discountThreshold)
        ? shippingMethod.discountedFee : shippingMethod.feePerCase;
  const shippingCost = quantity * shippingPerCase;
  const price = priceData.basePrice - discount + shippingCost;
  return price;
}
```

O próximo parâmetro da lista é shippingMethod. Deixo esse como está porque ele não é usado pelo código da primeira fase.

Depois disso, tenho quantity. Ele é usado pela primeira fase, mas não é criado por ela, portanto eu poderia deixá-lo na lista de parâmetros. No entanto, minha preferência, em geral, é mover o máximo de parâmetros possíveis para a estrutura de dados intermediária.

```
function priceOrder(product, quantity, shippingMethod) {
  const basePrice = product.basePrice * quantity;
  const discount = Math.max(quantity - product.discountThreshold, 0)
        * product.basePrice * product.discountRate;
  const priceData = {basePrice: basePrice, quantity: quantity};
  const price = applyShipping(priceData, shippingMethod, ~~quantity,~~ discount);
  return price;
```

```
}
function applyShipping(priceData, shippingMethod, quantity, discount) {
  const shippingPerCase = (priceData.basePrice > shippingMethod.discountThreshold)
        ? shippingMethod.discountedFee : shippingMethod.feePerCase;
  const shippingCost = priceData.quantity * shippingPerCase;
  const price = priceData.basePrice - discount + shippingCost;
  return price;
}
```

Faço o mesmo com discount.

```
function priceOrder(product, quantity, shippingMethod) {
  const basePrice = product.basePrice * quantity;
  const discount = Math.max(quantity - product.discountThreshold, 0)
        * product.basePrice * product.discountRate;
  const priceData = {basePrice: basePrice, quantity: quantity, discount:discount};
  const price = applyShipping(priceData, shippingMethod, discount);
  return price;
}
function applyShipping(priceData, shippingMethod, discount) {
  const shippingPerCase = (priceData.basePrice > shippingMethod.discountThreshold)
        ? shippingMethod.discountedFee : shippingMethod.feePerCase;
  const shippingCost = priceData.quantity * shippingPerCase;
  const price = priceData.basePrice - priceData.discount + shippingCost;
  return price;
}
```

Depois de ter analisado todos os parâmetros da função, tenho a estrutura de dados intermediária totalmente formada. Desse modo, posso extrair o código da primeira fase para a sua própria função, devolvendo esses dados.

```
function priceOrder(product, quantity, shippingMethod) {
  const priceData = calculatePricingData(product, quantity);
  const price = applyShipping(priceData, shippingMethod);
  return price;
}
function calculatePricingData(product, quantity) {
  const basePrice = product.basePrice * quantity;
  const discount = Math.max(quantity - product.discountThreshold, 0)
        * product.basePrice * product.discountRate;
  return {basePrice: basePrice, quantity: quantity, discount:discount};
}
function applyShipping(priceData, shippingMethod) {
  const shippingPerCase = (priceData.basePrice > shippingMethod.discountThreshold)
        ? shippingMethod.discountedFee : shippingMethod.feePerCase;
  const shippingCost = priceData.quantity * shippingPerCase;
```

```
  const price = priceData.basePrice - priceData.discount + shippingCost;
  return price;
}
```

Não consigo resistir e organizo as constantes finais.

```
function priceOrder(product, quantity, shippingMethod) {
  const priceData = calculatePricingData(product, quantity);
  return applyShipping(priceData, shippingMethod);
}

function calculatePricingData(product, quantity) {
  const basePrice = product.basePrice * quantity;
  const discount = Math.max(quantity - product.discountThreshold, 0)
        * product.basePrice * product.discountRate;
  return {basePrice: basePrice, quantity: quantity, discount:discount};
}
function applyShipping(priceData, shippingMethod) {
  const shippingPerCase = (priceData.basePrice > shippingMethod.discountThreshold)
        ? shippingMethod.discountedFee : shippingMethod.feePerCase;
  const shippingCost = priceData.quantity * shippingPerCase;
  return priceData.basePrice - priceData.discount + shippingCost;
}
```

CAPÍTULO 7
Encapsulamento

Talvez os critérios mais importantes a serem usados na decomposição de módulos seja identificar os segredos que os módulos devem ocultar do restante do sistema [Parnas]. As estruturas de dados são os segredos mais comuns, e posso ocultá-las encapsulando-as com *Encapsular registro (Encapsulate Record)* e *Encapsular coleção (Encapsulate Collection)*. Até mesmo valores primitivos podem ser encapsulados com *Substituir primitivo por objeto (Replace Primitive with Object)* – a dimensão das vantagens secundárias consequentes disso muitas vezes surpreende as pessoas. Com frequência, as variáveis temporárias atrapalham a refatoração – devo garantir que elas sejam calculadas na ordem correta e que seus valores estejam disponíveis a outras partes do código que precisem delas. Usar *Substituir variável temporária por consulta (Replace Temp with Query)* constitui uma ótima ajuda nesse caso, particularmente quando dividimos uma função que seja muito longa.

As classes foram concebidas para ocultar informações. No capítulo anterior, descrevi uma forma de criá-las com *Combinar funções em classe (Combine Functions into Class)*. As operações comuns de extração/internalização também se aplicam às classes com *Extrair classe (Extract Class)* e *Internalizar classe (Inline Class)*.

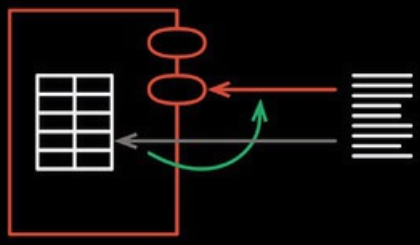
Além de ocultar informações internas, em geral as classes são convenientes para ocultar conexões entre elas, e isso pode ser feito com *Ocultar delegação (Hide Delegate)*. Entretanto, ocultar demais resulta em interfaces inchadas, portanto preciso também de sua inversa: *Remover intermediário (Remove Middle Man)*.

Classes e módulos são as formas mais amplas de encapsulamento, mas funções também encapsulam a sua implementação. Às vezes, posso precisar de uma mudança completa em um algoritmo, e isso pode ser feito encapsulando-o em uma função usando *Extrair função (Extract Function)* e

aplicando *Substituir algoritmo (Substitute Algorithm)*.

Encapsular registro (Encapsulate Record)

anteriormente: *Substituir registro por classe de dados* (Replace Record with Data Class)



```
organization = {name: "Acme Gooseberries", country: "GB"};
```

```
class Organization {
  constructor(data) {
    this._name = data.name;
    this._country = data.country;
  }
  get name() {return this._name;}
  set name(arg) {this._name = arg;}
  get country() {return this._country;}
  set country(arg) {this._country = arg;}
}
```

Motivação

Estruturas de registro são um recurso comum nas linguagens de programação. Elas oferecem um modo intuitivo para agrupar dados relacionados, permitindo que eu passe unidades de dados significativas, em vez de conjuntos desconectados. No entanto, estruturas de registro simples têm desvantagens. A mais irritante delas é o fato de me forçarem a separar claramente o que está armazenado no registro dos valores calculados. Considere a noção de um intervalo inclusivo de inteiros. Posso armazenar

esse intervalo como {start: 1, end:5} ou {start: 1, length:5} (ou até mesmo {end: 5, length:5}, se eu quiser evidenciar o meu antagonismo). No entanto, independentemente do que eu armazenar, quero saber o início (start), o fim (end) e o tamanho (length).

É por isso que, em geral, prefiro objetos aos registros para dados mutáveis. Com os objetos, posso ocultar o que é armazenado e disponibilizar métodos para todos os três valores. O usuário do objeto não precisa saber nem se importar com o que é armazenado e o que é calculado. Esse encapsulamento também ajuda caso eu queira renomear itens: posso renomear o campo enquanto disponibilizo métodos tanto para o nome antigo quanto para o nome novo, atualizando gradualmente as chamadas até que todas estejam atualizadas.

Acabei de dizer que prefiro objetos para dados *mutáveis*. Se eu tiver um valor imutável, posso ter simplesmente os três valores em meu registro, usando um passo de enriquecimento, se for necessário. De modo semelhante, é fácil copiar o campo ao renomear.

Posso ter dois tipos de estruturas de registro: aquelas em que declaro os nomes oficiais dos campos e aquelas que me permitem usar o que eu quiser. As últimas em geral são implementadas com uma classe de biblioteca cujo nome é algo como hash, mapa (map), hashmap, dicionário (dictionary) ou array associativo (associative array). Muitas linguagens oferecem uma sintaxe apropriada para criar hashmaps, o que os torna convenientes em muitos cenários de programação. A desvantagem de usá-los é que eles não são explícitos quanto a seus campos. O único modo pelo qual posso saber se usam início/fim ou início/tamanho é observando o local em que são criados e usados. Isso não será um problema se eles forem usados somente em uma pequena seção de um programa; porém, quanto mais amplo for seu escopo de uso, mais problemas terei em consequência de sua estrutura implícita. Eu poderia refatorar esses registros implícitos e transformá-los em explícitos – mas, se tiver de fazer isso, prefiro transformá-los em classes.

É comum passar estruturas aninhadas de listas e hashmaps, as quais, com frequência, são serializadas em formatos como JSON e XML. Estruturas como essas também podem ser encapsuladas; isso ajudará se seus formatos mudarem depois ou se eu estiver preocupado com atualizações nos dados que sejam difíceis de monitorar.

Procedimento

- Use *Encapsular variável (Encapsulate Variable)* na variável que armazena o registro.

Dê nomes fáceis de procurar às funções que encapsulam o registro.

- Substitua o conteúdo da variável por uma classe simples que encapsule o registro. Defina um método de acesso nessa classe que devolva o registro bruto. Modifique as funções que encapsulam a variável para que usem esse método de acesso.
- Teste.
- Disponibilize novas funções que devolvam o objeto no lugar do registro bruto.
- Para cada usuário do registro, substitua o uso de uma função que devolve o registro por uma função que devolva o objeto. Use um método de acesso no objeto para obter o dado de um campo, criando esse método de acesso se necessário. Teste após cada mudança.

Se for um registro complexo, por exemplo, um registro com uma estrutura aninhada, concentre-se nos clientes que atualizam os dados antes. Considere devolver uma cópia ou um proxy somente de leitura dos dados para os clientes que apenas leiam os dados.

- Remova o método de acesso para os dados brutos da classe e as funções com nomes fáceis de procurar, que devolviam o registro bruto.
- Teste.
- Se os campos do registro forem, eles mesmos, estruturas, considere usar *Encapsular registro (Encapsulate Record)* e *Encapsular coleção (Encapsulate Collection)* recursivamente.

Exemplo

Começarei com uma constante amplamente usada em um programa.

```
const organization = {name: "Acme Gooseberries", country: "GB"};
```

Esse é um objeto JavaScript sendo usado como uma estrutura de registro por várias partes do programa, com acessos como estes:

```
result += `<h1>${organization.name}</h1>`;
```

e:

```
organization.name = newName;
```

O primeiro passo é um simples *Encapsular variável (Encapsulate Variable)*.

```
function getRawDataOfOrganization() {return organization;}
```

exemplo de leitura…

```
result += `<h1>${getRawDataOfOrganization().name}</h1>`;
```

exemplo de escrita…

```
getRawDataOfOrganization().name = newName;
```

Não é exatamente um *Encapsular variável (Encapsulate Variable) padrão, pois dei ao* getter um nome propositalmente escolhido para ser, ao mesmo tempo, feio e fácil de procurar. Isso porque minha intenção é que seu tempo de vida seja breve.

Encapsular um registro significa ir além da variável em si; quero controlar como ela é manipulada. Posso fazer isso substituindo o registro por uma classe.

class Organization…

```
class Organization {
  constructor(data) {
    this._data = data;
  }
}
```

nível mais alto

```
const organization = new Organization({name: "Acme Gooseberries", country: "GB"});
function getRawDataOfOrganization() {return organization._data;}
function getOrganization() {return organization;}
```

Agora que tenho um objeto definido, começo observando os usuários do registro. Qualquer um que atualize o registro deve ser substituído por um setter.

class Organization…

```
set name(aString) {this._data.name = aString;}
```

cliente…

```
getOrganization().name = newName;
```

De modo semelhante, substituo qualquer leitura pelo getter apropriado.

class Organization…

```
get name() {return this._data.name;}
```

cliente…

```
result += `<h1>${getOrganization().name}</h1>`;
```

Depois de fazer isso, posso cumprir minha ameaça de dar uma vida breve à função com o nome feio.

```
function getRawDataOfOrganization() {return organization._data;}
function getOrganization() {return organization;}
```

Também me sinto inclinado a desdobrar o campo _data diretamente no objeto.

```
class Organization {
  constructor(data) {
    this._name = data.name;
    this._country = data.country;
  }
  get name() {return this._name;}
  set name(aString) {this._name = aString;}
  get country() {return this._country;}
  set country(aCountryCode) {this._country = aCountryCode;}
}
```

Isso tem a vantagem de romper a ligação com o registro de dados de entrada. Pode ser conveniente se houver uma referência a ele em outros lugares, causando uma quebra no encapsulamento. Se eu não desdobrar os dados em campos individuais, copiar _data quando de sua atribuição seria uma atitude inteligente.

Exemplo: encapsulando um registro aninhado

O exemplo anterior usou um registro simples, mas o que devo fazer se houver dados aninhados com mais níveis de profundidade, por exemplo, provenientes de um documento JSON? Os passos principais da refatoração ainda se aplicam, e tenho de ser igualmente cuidadoso com as atualizações, mas há algumas opções quanto às leituras.

Como exemplo, eis alguns dados um pouco mais aninhados: uma coleção de clientes, mantidos em um hashmap indexado pelo ID do cliente.

```
"1920": {
  name: "martin",
  id: "1920",
  usages: {
    "2016": {
      "1": 50,
      "2": 55,
      // meses restantes do ano
    },
    "2015": {
      "1": 70,
```

```
        "2": 63,
        // meses restantes do ano
      }
    }
  },
  "38673": {
    name: "neal",
    id: "38673",
    // mais clientes em formato semelhante
```

Com mais dados aninhados, as leituras e as escritas podem acessar a estrutura de dados em níveis mais profundos.

exemplo de atualização…

```
customerData[customerID].usages[year][month] = amount;
```

exemplo de leitura…

```
function compareUsage (customerID, laterYear, month) {
  const later = customerData[customerID].usages[laterYear][month];
  const earlier = customerData[customerID].usages[laterYear - 1][month];
  return {laterAmount: later, change: later - earlier};
}
```

Para encapsular esses dados, também começo usando *Encapsular variável (Encapsulate Variable)*.

```
function getRawDataOfCustomers() {return customerData;}
function setRawDataOfCustomers(arg) {customerData = arg;}
```

exemplo de atualização…

```
getRawDataOfCustomers()[customerID].usages[year][month] = amount;
```

exemplo de leitura…

```
function compareUsage (customerID, laterYear, month) {
  const later = getRawDataOfCustomers()[customerID].usages[laterYear][month];
  const earlier = getRawDataOfCustomers()[customerID].usages[laterYear - 1][month];
  return {laterAmount: later, change: later - earlier};
}
```

Então crio uma classe para a estrutura de dados como um todo.

```
class CustomerData {
  constructor(data) {
    this._data = data;
  }
}
```

nível mais alto…

```
function getCustomerData() {return customerData;}
function getRawDataOfCustomers() {return customerData._data;}
function setRawDataOfCustomers(arg) {customerData = new CustomerData(arg);}
```

A parte mais importante para tratar são as atualizações. Então, embora eu observe todos os pontos de chamada de `getRawDataOfCustomers`, mantenho o foco naqueles em que os dados são modificados. Para recordar, eis a atualização novamente:

exemplo de atualização…

```
getRawDataOfCustomers()[customerID].usages[year][month] = amount;
```

O procedimento geral determina agora que devo devolver o cliente completo e usar um método de acesso, criando um se for necessário. Não tenho um setter no cliente para essa atualização, e esta acessa a estrutura em profundidade. Assim, para criar um, começo usando *Extrair função (Extract Function)* no código que faz esse acesso profundo na estrutura de dados.

exemplo de atualização…

```
setUsage(customerID, year, month, amount);
```

nível mais alto…

```
function setUsage(customerID, year, month, amount) {
  getRawDataOfCustomers()[customerID].usages[year][month] = amount;
}
```

Então uso *Mover função (Move Function)* a fim de movê-la para a nova classe de dados de cliente.

exemplo de atualização…

```
getCustomerData().setUsage(customerID, year, month, amount);
```

class CustomerData…

```
setUsage(customerID, year, month, amount) {
  this._data[customerID].usages[year][month] = amount;
}
```

Ao trabalhar com uma estrutura de dados grande, gosto de me concentrar nas atualizações. Deixá-las visíveis e agrupadas em um só lugar é a parte mais importante do encapsulamento.

Em algum momento, vou achar que cuidei de todas – mas como posso ter certeza? Há algumas maneiras de verificar isso. Uma delas é modificar `getRawDataOfCustomers` para que devolva uma cópia profunda (deep copy) dos dados; se minha cobertura de testes for boa, um dos testes deverá falhar se eu

tiver me esquecido de alguma modificação.

nível mais alto…

```
function getCustomerData() {return customerData;}
function getRawDataOfCustomers() {return customerData.rawData;}
function setRawDataOfCustomers(arg) {customerData = new CustomerData(arg);}
```

class CustomerData…

```
  get rawData() {
    return _.cloneDeep(this._data);
  }
```

Estou usando a biblioteca lodash para fazer uma cópia profunda.

Outra abordagem consiste em devolver um proxy somente de leitura para a estrutura de dados. Um proxy como esse poderia lançar uma exceção se o código do cliente tentar modificar o objeto subjacente. Algumas linguagens facilitam essa tarefa, mas é complicado em JavaScript, portanto deixarei esse código como um exercício para o leitor. Eu poderia também fazer uma cópia e congelá-la recursivamente para detectar qualquer modificação.

Lidar com as atualizações é importante, mas e quanto às leituras? Nesse caso, há algumas opções.

A primeira delas é fazer o mesmo que fiz para os setters. Extraia todas as leituras em suas próprias funções e mova-as para a classe de dados de cliente.

class CustomerData…

```
usage(customerID, year, month) {
  return this._data[customerID].usages[year][month];
}
```

nível mais alto…

```
function compareUsage (customerID, laterYear, month) {
  const later = getCustomerData().usage(customerID, laterYear, month);
  const earlier = getCustomerData().usage(customerID, laterYear - 1, month);
  return {laterAmount: later, change: later - earlier};
}
```

A parte interessante dessa abordagem é que ela proporciona uma API explícita para customerData, a qual captura todos os usos que lhe são feitos. Posso olhar para a classe e ver todos os usos que se fazem dos dados. Contudo, pode haver muito código para vários casos especiais. As linguagens modernas oferecem bons recursos para explorar uma estrutura de dados lista-e-hash [mf-lh], portanto é conveniente oferecer uma estrutura de dados como

essa para que os clientes trabalhem com ela.

Se o cliente quiser uma estrutura de dados, posso apenas lhe passar os dados propriamente ditos. O problema com isso, porém, é que não há meios de evitar que os clientes modifiquem os dados diretamente, o que acaba com todo o propósito de encapsular as atualizações em funções. Consequentemente, a tarefa mais simples a ser feita é fornecer uma cópia dos dados subjacentes, usando o método rawData que escrevi antes.

class CustomerData…

```
get rawData() {
  return _.cloneDeep(this._data);
}
```

nível mais alto…

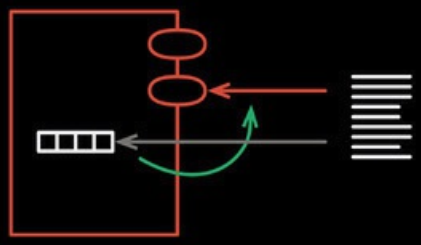
```
function compareUsage (customerID, laterYear, month) {
  const later = getCustomerData().rawData[customerID].usages[laterYear][month];
  const earlier = getCustomerData().rawData[customerID].usages[laterYear - 1][month];
  return {laterAmount: later, change: later - earlier};
}
```

Embora seja simples, há desvantagens nisso. O problema mais evidente é o custo de copiar uma estrutura de dados grande, o que pode acabar resultando em um problema de desempenho. Como em qualquer situação como essa, porém, o custo para o desempenho pode ser aceitável – eu avaliaria seu impacto antes de começar a me preocupar com ele. Também pode haver confusão se os clientes estiverem esperando modificar os dados originais ao modificar os dados copiados. Nesses casos, um proxy somente de leitura ou um congelamento dos dados copiados pode gerar um erro conveniente caso isso ocorra.

Outra opção é mais trabalhosa, mas oferece o controle máximo: aplicar *Encapsular registro (Encapsulate Record)* recursivamente. Com ela, transformo o registro do cliente em uma classe própria, aplico *Encapsular coleção (Encapsulate Collection)* nos usos e crio uma classe de uso. Posso então garantir um controle das atualizações usando métodos de acesso, talvez aplicando *Mudar referência para valor (Change Reference to Value)* nos objetos de uso. Contudo, isso pode significar muito trabalho para uma estrutura de dados grande – e não realmente necessário, se eu não acessar muito a estrutura de dados. Às vezes, uma mistura criteriosa de getters e novas classes pode funcionar, usando um getter para acessar a estrutura em profundidade, porém devolvendo um objeto que encapsule a estrutura, em

vez de devolver os dados não encapsulados. Escrevi a respeito desse tipo de situação em um artigo chamado “Refatorando código para carregar um documento” (Refactoring Code to Load a Document) [mf-ref-doc].

Encapsular coleção (Encapsulate Collection)



```
class Person {
  get courses() {return this._courses;}
  set courses(aList) {this._courses =
aList;}
```

```
class Person {
  get courses() {return
this._courses.slice();}
  addCourse(aCourse) { ... }
  removeCourse(aCourse) { ... }
```

Motivação

Gosto de encapsular qualquer dado mutável em meus programas. Isso facilita ver quando e como as estruturas de dados são modificadas, o que, por sua vez, facilita modificar essas estruturas de dados quando necessário. Com frequência, o encapsulamento é incentivado, particularmente por desenvolvedores que usam orientação a objetos, mas um erro comum ocorre quando trabalhamos com coleções. O acesso a uma variável de coleção pode estar encapsulado, mas, se o getter devolver a própria coleção, os membros dessa coleção poderão ser alterados sem que a classe que faz o encapsulamento possa intervir.

Para evitar isso, disponibilizo métodos para modificar a coleção – em geral, `add` e `remove` – na própria classe. Desse modo, as alterações na coleção passam pela classe responsável, dando-me a oportunidade de alterar essas

modificações à medida que o programa evoluir.

Se a equipe tiver o hábito de não modificar coleções fora do módulo original, somente disponibilizar esses métodos pode ser suficiente. Todavia, contar com esses hábitos, em geral, não é uma atitude sábia; um erro pode resultar em bugs difíceis de identificar no futuro. Uma abordagem mais conveniente é garantir que o getter da coleção não devolva a coleção bruta, de modo que os clientes não possam alterá-la acidentalmente.

Uma maneira de impedir a modificação da coleção subjacente é jamais devolver um valor de coleção. Nessa abordagem, qualquer uso de um campo de coleção é feito com métodos específicos da classe responsável, substituindo `aCustomer.orders.size` por `aCustomer.numberOfOrders`. Não concordo com essa abordagem. As linguagens modernas têm classes de coleção ricas, com interfaces padronizadas, que podem ser combinadas de formas convenientes, como Pipelines de Coleções (Collection Pipelines) [mf-cp]. Criar métodos especiais para lidar com esse tipo de funcionalidade implica a adição de muito código extra e dificulta usar facilmente o recurso de composição das operações de coleção.

Outra forma é permitir algum tipo de acesso apenas de leitura a uma coleção. Java, por exemplo, facilita devolver um proxy somente de leitura para a coleção. Um proxy como esse encaminha todas as leituras para a coleção subjacente, porém bloqueia todas as escritas – no caso de Java, lançando uma exceção. Uma alternativa semelhante é usada por bibliotecas que baseiam sua composição de coleções em algum tipo de iterador ou objeto enumerável – desde que esse iterador não modifique a coleção subjacente.

Provavelmente a abordagem mais comum é disponibilizar um método de leitura para a coleção, mas fazê-la devolver uma cópia da coleção subjacente. Desse modo, qualquer modificação na cópia não afetará a coleção encapsulada. Isso pode causar certa confusão caso os programadores esperem que a coleção devolvida modifique o campo original – porém, em muitas bases de código, eles estão acostumados com o fato de os getters de coleção devolverem cópias. Se a coleção for enorme, pode haver problema de desempenho – a maioria das listas, porém, não é tão grande assim, portanto as regras gerais sobre desempenho devem se aplicar (veja a seção *Refatoração e desempenho* do Capítulo 2 na *página 87*).

Outra diferença entre usar um proxy e uma cópia é o fato de uma modificação nos dados originais ser visível no proxy, mas não em uma cópia. Isso não será um problema na maioria das vezes porque as listas acessadas

dessa forma em geral são mantidas somente por pouco tempo.

O que importa nesse caso é a consistência em uma base de código. Use apenas um procedimento para que todos se acostumem com o seu comportamento e esperem por ele quando chamarem qualquer função de acesso a uma coleção.

Procedimento

- Aplique *Encapsular variável (Encapsulate Variable)* se a referência à coleção ainda não estiver encapsulada.
- Acrescente funções para adicionar e remover elementos da coleção.

Caso haja um setter para a coleção, use *Remover método de escrita (Remove Setting Method)* se possível. Se não for, faça com que uma cópia da coleção seja usada.

- Execute verificações estáticas.
- Encontre todas as referências à coleção. Se alguém chamar modificadores na coleção, altere-os para que usem as novas funções de adição/remoção. Teste após cada mudança.
- Modifique o getter da coleção de modo que devolva uma visão protegida dela, usando um proxy somente de leitura ou uma cópia.
- Teste.

Exemplo

Começarei com uma classe pessoa que tenha um campo para uma lista de cursos.

class Person…

```
constructor (name) {
  this._name = name;
  this._courses = [];
}
get name() {return this._name;}
get courses() {return this._courses;}
set courses(aList) {this._courses = aList;}
```

class Course…

```
constructor(name, isAdvanced) {
  this._name = name;
```

```
    this._isAdvanced = isAdvanced;
  }
  get name() {return this._name;}
  get isAdvanced() {return this._isAdvanced;}
```

Os clientes usam a coleção de cursos para obter informações sobre eles.

```
numAdvancedCourses = aPerson.courses
  .filter(c => c.isAdvanced)
  .length
;
```

Um desenvolvedor ingênuo diria que essa classe tem um encapsulamento de dados apropriado: afinal de contas, cada campo está protegido por métodos de acesso. Entretanto, eu argumentaria que a lista de cursos não está devidamente encapsulada. Certamente, qualquer um que atualize os cursos com um único valor tem um controle apropriado por meio do setter:

código do cliente…

```
const basicCourseNames = readBasicCourseNames(filename);
aPerson.courses = basicCourseNames.map(name => new Course(name, false));
```

Porém, os clientes podem achar mais fácil atualizar a lista de cursos diretamente.

código do cliente…

```
for(const name of readBasicCourseNames(filename)) {
  aPerson.courses.push(new Course(name, false));
}
```

Isso viola o encapsulamento porque a classe pessoa não é capaz de manter o controle quando a lista é atualizada dessa maneira. Embora a referência ao campo esteja encapsulada, o conteúdo do campo não está.

Começarei criando um encapsulamento apropriado, acrescentando métodos na classe pessoa que permitam a um cliente adicionar e remover cursos individuais.

class Person…

```
addCourse(aCourse) {
  this._courses.push(aCourse);
}
removeCourse(aCourse, fnIfAbsent = () => {throw new RangeError();}) {
  const index = this._courses.indexOf(aCourse);
  if (index === -1) fnIfAbsent();
  else this._courses.splice(index, 1);
}
```

Em uma remoção, tenho de decidir o que fazer caso um cliente peça para remover um elemento que não esteja na coleção. Posso ignorar ou gerar um erro. Nesse código, gero um erro por padrão, mas dou uma oportunidade aos que fazem a chamada para que façam algo diferente caso queiram.

Então modifico qualquer código que chame diretamente os modificadores da coleção para que usem os novos métodos.

código do cliente…

```
for(const name of readBasicCourseNames(filename)) {
  aPerson.addCourse(new Course(name, false));
}
```

Com métodos de adição e de remoção individuais, em geral não há necessidade de ter setCourses; nesse caso, usarei *Remover método de escrita (Remove Setting Method)*. Caso a API por algum motivo necessite de um método de escrita, certifico-me de que ela coloque uma cópia da coleção no campo.

class Person…

```
set courses(aList) {this._courses = aList.slice();}
```

Tudo isso permite aos clientes usar o tipo correto de métodos de modificação, mas prefiro garantir que ninguém modifique a lista sem que usem esses métodos. Posso fazer isso fornecendo uma cópia.

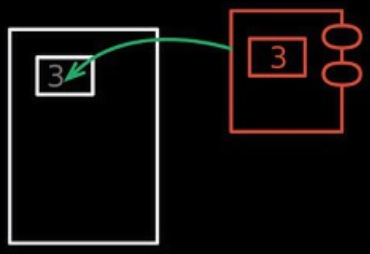
class Person…

```
get courses() {return this._courses.slice();}
```

Em geral, acho que ser moderadamente paranoico sobre coleções é uma atitude sábia, e prefiro copiá-las desnecessariamente a depurar erros em consequência de modificações inesperadas. As modificações nem sempre são óbvias; por exemplo, ordenar um array em JavaScript modifica o original, enquanto muitas linguagens têm como padrão fazer uma cópia para uma operação que modifique uma coleção. Qualquer classe que seja responsável por administrar uma coleção sempre deve fornecer cópias – mas tenho também o hábito de criar uma cópia caso eu faça algo que possa ser responsável por alterar uma coleção.

Substituir primitivo por objeto (Replace Primitive with Object)

anteriormente: *Substituir dado por objeto* (Replace Data Value with Object)
anteriormente: *Substituir código de tipos por classe (Replace Type Code with Class)*




```
orders.filter(o => "high" ===
o.priority
                        || "rush" ===
o.priority);
```

```
orders.filter(o => o.priority.higherThan(new
Priority("normal")))
```

Motivação

Com frequência, nas etapas iniciais do desenvolvimento, você toma decisões sobre representar fatos simples como dados simples, por exemplo, como números ou strings. À medida que o desenvolvimento avança, esses itens simples não serão mais tão simples. Um número de telefone pode ser representado como uma string por um tempo; mais tarde, porém, poderá exigir um comportamento especial para formatação, extração do código de área, e tarefas afins. Esse tipo de lógica pode acabar rapidamente sendo duplicado na base de código, exigindo mais esforços sempre que precisar ser utilizado.

Assim que percebo que quero fazer algo além de uma simples exibição da porção de dados, gosto de criar uma nova classe para ela. Em um primeiro momento, uma classe como essa faz somente pouco mais do que encapsular o primitivo – mas, depois que eu tiver essa classe, terei um local para colocar comportamentos específicos às suas necessidades. Esses pequenos valores começam muito humildemente, mas, depois de nutridos, poderão crescer e se transformar em ferramentas úteis. Embora não pareçam muito, acho que seus

efeitos em uma base de código podem ser surpreendentemente grandes. De fato, muitos desenvolvedores experientes consideram essa refatoração uma das mais importantes do kit de ferramentas – mesmo que, muitas vezes, ela pareça contraintuitiva para um novo programador.

Procedimento

- Aplique *Encapsular variável (Encapsulate Variable)* se ela ainda não estiver encapsulada.
- Crie uma classe simples de valor para o dado. Ela deve aceitar o valor existente em seu construtor e disponibilizar um getter para esse valor.
- Execute verificações estáticas.
- Modifique o setter a fim de criar uma nova instância da classe de valor e armazená-la no campo, alterando o tipo do campo, se estiver presente.
- Modifique o getter para que devolva o resultado da chamada do getter da nova classe.
- Teste.
- Considere usar *Renomear função (Rename Function)* nos métodos de acesso originais para refletir melhor o que eles fazem.
- Considere deixar mais claro o papel do novo objeto como um objeto de valor ou de referência aplicando *Mudar referência para valor (Change Reference to Value)* ou *Mudar valor para referência (Change Value to Reference)*.

Exemplo

Começo com uma classe de pedido (order) simples que lê seus dados de uma estrutura de registro simples. Uma de suas propriedades é uma prioridade (priority), lida como uma string simples.

class Order…

```
constructor(data) {
  this.priority = data.priority;
  // outras inicializações
```

Alguns códigos de cliente usam o campo da seguinte maneira:

cliente…

```
highPriorityCount = orders.filter(o => "high" === o.priority
```

```
                || "rush" === o.priority)
        .length;
```

Sempre que estou lidando com um valor, minha primeira atitude é usar *Encapsular variável (Encapsulate Variable)*.

class Order…

```
get priority() {return this._priority;}
set priority(aString) {this._priority = aString;}
```

A linha do construtor que inicializa a prioridade agora usará o setter que defini nesse código.

Com isso, o campo é autoencapsulado, de modo que posso preservar seu uso atual, ao mesmo tempo em que manipulo o dado propriamente dito.

Crio uma classe de valor simples para a prioridade. Ela tem um construtor para o valor e uma função de conversão que devolve uma string.

```
class Priority {
  constructor(value) {this._value = value;}
  toString() {return this._value;}
}
```

Prefiro usar uma função de conversão (toString) a um getter (value) nesse caso. Para os clientes da classe, pedir a representação em string deve se assemelhar mais a uma conversão do que a uma leitura de propriedade.

Em seguida, modifico os métodos de acesso para que usem essa nova classe.

class Order…

```
get priority() {return this._priority.toString();}
set priority(aString) {this._priority = new Priority(aString);}
```

Agora que tenho uma classe de prioridade, acho confuso o getter que está atualmente na classe de pedido. Ele não devolve a prioridade – mas uma string que descreve a prioridade. Meu próximo passo é usar *Renomear função (Rename Function)*.

class Order…

```
get priorityString() {return this._priority.toString();}
set priority(aString) {this._priority = new Priority(aString);}
```

cliente…

```
highPriorityCount = orders.filter(o => "high" === o.priorityString
                || "rush" === o.priorityString)
```

```
                    .length;
```

Nesse caso, fico satisfeito em preservar o nome do setter. O nome do argumento comunica o que ele espera.

Terminei a refatoração formal. No entanto, quando olho para quem usa a prioridade, considero se eles devem usar a classe de prioridade. Como resultado, disponibilizo um getter no pedido, que disponibiliza diretamente o novo objeto de prioridade.

class Order…

```
get priority() {return this._priority;}
get priorityString() {return this._priority.toString();}
set priority(aString) {this._priority = new Priority(aString);}
```

cliente…

```
highPriorityCount = orders.filter(o => "high" === o.priority.toString()
                                    || "rush" === o.priority.toString())
                    .length;
```

À medida que a classe de prioridade se torna útil em outros lugares, permitirei aos clientes do pedido usar o setter com uma instância da prioridade, o que é feito modificando o construtor da prioridade.

class Priority…

```
constructor(value) {
  if (value instanceof Priority) return value;
  this._value = value;
}
```

A questão principal em tudo isso é que, agora, minha nova classe de prioridade pode ser útil como um local para novos comportamentos – sejam novos no código ou transferidos de outros lugares. Eis um código simples para acrescentar validação de valores de prioridade e uma lógica de comparação:

class Priority…

```
constructor(value) {
  if (value instanceof Priority) return value;
  if (Priority.legalValues().includes(value))
    this._value = value;
  else
    throw new Error(`<${value}> is invalid for Priority`);
}
toString() {return this._value;}
```

```
  get _index() {return Priority.legalValues().findIndex(s => s === this._value);}
  static legalValues() {return ['low', 'normal', 'high', 'rush'];}

  equals(other) {return this._index === other._index;}
  higherThan(other) {return this._index > other._index;}
  lowerThan(other) {return this._index < other._index;}
```

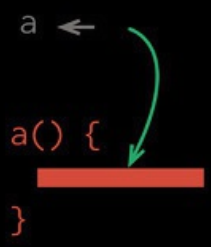
Enquanto faço isso, decido que uma prioridade deve ser um objeto de valor, portanto disponibilizo um método de igualdade e garanto que ele seja imutável.

Agora que acrescentei esse comportamento, posso deixar o código do cliente mais significativo:

cliente…

```
  highPriorityCount = orders.filter(o => o.priority.higherThan(new Priority("normal")))
                            .length;
```

Substituir variável temporária por consulta (Replace Temp with Query)




```
const basePrice = this._quantity *
this._itemPrice;
if (basePrice > 1000)
  return basePrice * 0.95;
else
  return basePrice * 0.98;
```

```
get basePrice() {this._quantity *
this._itemPrice;}

...

if (this.basePrice > 1000)
```

```
    return this.basePrice * 0.95;
  else
    return this.basePrice * 0.98;
```

Motivação

Um uso para as variáveis temporárias é capturar o valor de um código a fim de referenciá-lo mais tarde em uma função. Usar uma variável temporária me permite referenciar o valor, ao mesmo tempo que explica seu significado e evita repetir o código que a calcula. No entanto, embora usar uma variável seja prático, muitas vezes pode valer a pena dar um passo além e usar uma função em seu lugar.

Se eu estiver trabalhando para dividir uma função grande, transformar variáveis em funções próprias facilita extrair partes da função, pois não precisarei mais passar variáveis para as funções extraídas. Colocar essa lógica em funções muitas vezes define uma fronteira mais clara entre a lógica extraída e a função original, ajudando-me a identificar e a evitar dependências inconvenientes e efeitos colaterais.

Usar funções no lugar de variáveis também me permite evitar duplicação da lógica de cálculo em funções semelhantes. Sempre que vejo variáveis calculadas do mesmo modo em lugares diferentes, procuro transformá-las em uma única função.

Essa refatoração funciona melhor se eu estiver dentro de uma classe, pois a classe oferece um contexto compartilhado para os métodos que estou extraindo. Fora de uma classe, estou sujeito a ter muitos parâmetros em uma função de nível mais alto, o que anulará boa parte das vantagens de usar uma função. Funções aninhadas podem evitar isso, mas limitam minha capacidade de compartilhar a lógica entre funções relacionadas.

Somente algumas variáveis temporárias são apropriadas para o uso de *Substituir variável temporária por consulta*. A variável deve ser calculada uma vez, e então deve ser apenas lida depois. No caso mais simples, isso significa que a variável recebe valor uma vez, mas também é possível haver várias atribuições em uma porção de código mais complicada – todas elas devem ser extraídas e colocadas na consulta. Além do mais, a lógica usada para calcular a variável deve gerar o mesmo resultado quando a variável for usada mais tarde – o que exclui as variáveis usadas como snapshots (imagens instantâneas) com nomes como oldAddress.

Procedimento

- Verifique se a variável é totalmente definida antes de ser usada, e certifique-se de que o código que a calcula não gere um valor diferente sempre que ela for usada.
- Se a variável não for somente de leitura, mas puder sê-lo, faça isso.
- Teste.
- Extraia a atribuição da variável em uma função.

Se a variável e a função não puderem compartilhar um nome, utilize um nome temporário para a função.

Garanta que a função extraída esteja livre de efeitos colaterais. Se não estiver, use *Separar consulta de modificador (Separate Query from Modifier)*.

- Teste.
- Use *Internalizar variável (Inline Variable)* para remover a variável temporária.

Exemplo

Eis uma classe simples:

class Order…

```
  constructor(quantity, item) {
    this._quantity = quantity;
    this._item = item;
  }

  get price() {
    var basePrice = this._quantity * this._item.price;
    var discountFactor = 0.98;
    if (basePrice > 1000) discountFactor -= 0.03;
    return basePrice * discountFactor;
  }
}
```

Quero substituir as variáveis temporárias basePrice e discountFactor por métodos.

Começando com basePrice, faço com que seja const e executo os testes. Essa é uma boa maneira de verificar se não me esqueci de nenhuma atribuição – é improvável em uma função pequena como essa, mas será comum se eu estiver lidando com um código maior.

class Order…

```
  constructor(quantity, item) {
    this._quantity = quantity;
    this._item = item;
  }

  get price() {
    const basePrice = this._quantity * this._item.price;
    var discountFactor = 0.98;
    if (basePrice > 1000) discountFactor -= 0.03;
    return basePrice * discountFactor;
  }
}
```

Então extraio o lado direito da atribuição para um método de leitura.

class Order…

```
get price() {
  const basePrice = this.basePrice;
  var discountFactor = 0.98;
  if (basePrice > 1000) discountFactor -= 0.03;
  return basePrice * discountFactor;
}
get basePrice() {
  return this._quantity * this._item.price;
}
```

Testo e aplico *Internalizar variável (Inline Variable)*.

class Order…

```
get price() {
  const basePrice = this.basePrice;
  var discountFactor = 0.98;
  if (this.basePrice > 1000) discountFactor -= 0.03;
  return this.basePrice * discountFactor;
}
```

Então repito os passos com discountFactor, usando *Extrair função (Extract Function)* antes.

class Order…

```
get price() {
  const discountFactor = this.discountFactor;
  return this.basePrice * discountFactor;
}
get discountFactor() {
```

```
    var discountFactor = 0.98;
    if (this.basePrice > 1000) discountFactor -= 0.03;
    return discountFactor;
  }
```

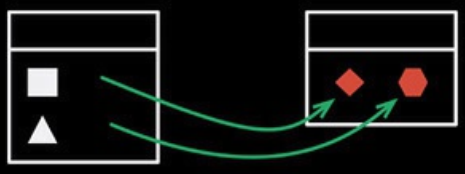
Nesse caso, preciso que a função extraída contenha as duas atribuições a discountFactor. Também posso definir a variável original como const.

Em seguida, eu a internalizo:

```
get price() {
  return this.basePrice * this.discountFactor;
}
```

Extrair classe (Extract Class)

inversa de: *Internalizar classe (Inline Class)*



```
class Person {
  get officeAreaCode() {return
this._officeAreaCode;}
  get officeNumber() {return this._officeNumber;}
```

```
class Person {
  get officeAreaCode() {return
this._telephoneNumber.areaCode;}
  get officeNumber() {return this._telephoneNumber.number;}
}
class TelephoneNumber {
  get areaCode() {return this._areaCode;}
  get number() {return this._number;}
}
```

Motivação

Você provavelmente já viu orientações dizendo que uma classe deve ser uma abstração nítida, deve lidar somente com algumas responsabilidades claras, e

assim por diante. Na prática, as classes crescem. Você acrescenta algumas operações aqui, um pouco de dados ali. Adiciona uma responsabilidade em uma classe achando que não vale a pena ter uma classe separada – mas, à medida que essa responsabilidade aumenta, a classe se torna complicada demais. Em breve, sua classe não será mais tão clara assim.

Pense em uma classe com muitos métodos e uma grande quantidade de dados. Uma classe grande demais para ser facilmente compreendida. Você deve considerar em que ponto ela pode ser separada – e separá-la. Um bom sinal é quando um subconjunto dos dados e um subconjunto dos métodos parecem formar um conjunto. Outros bons sinais são subconjuntos de dados que geralmente mudam juntos ou são particularmente dependentes uns dos outros. Um teste conveniente é perguntar a si mesmo o que aconteceria se você removesse uma porção dos dados ou um método. Quais outros campos e métodos deixariam de fazer sentido?

Um sinal que muitas vezes surge mais tarde no desenvolvimento é o modo como a classe é subtipada. Você poderá perceber que a subtipagem afeta somente alguns recursos ou que alguns recursos devem ser subtipados de uma forma, enquanto outros, de forma diferente.

Procedimento

- Decida como separar as responsabilidades da classe.
- Crie uma nova classe-filha para expressar as responsabilidades da classe separada.

Se as responsabilidades da classe-pai original não corresponderem mais ao seu nome, renomeie a classe-pai.

- Crie uma instância da classe-filha quando construir a classe-pai e adicione uma ligação da classe-pai para a classe-filha.
- Use *Mover campo (Move Field)* em cada campo que quiser mover. Teste após cada mudança.
- Use *Mover função (Move Function)* para mover métodos para a nova classe-filha. Comece pelos métodos de nível mais baixo (aqueles que são chamados em vez de fazer chamadas). Teste após cada mudança.
- Revise as interfaces das duas classes, remova os métodos desnecessários, altere nomes para que estejam mais apropriados às novas circunstâncias.
- Decida se deve expor a nova classe-filha. Em caso afirmativo, considere

aplicar *Mudar referência para valor (Change Reference to Value)* na nova classe-filha.

Exemplo

Começarei com uma classe simples de pessoa:

class Person…

```
get name() {return this._name;}
set name(arg) {this._name = arg;}
get telephoneNumber() {return `(${this.officeAreaCode}) ${this.officeNumber}`;}
get officeAreaCode() {return this._officeAreaCode;}
set officeAreaCode(arg) {this._officeAreaCode = arg;}
get officeNumber() {return this._officeNumber;}
set officeNumber(arg) {this._officeNumber = arg;}
```

Nesse caso, posso separar o comportamento do número de telefone em sua própria classe. Começo definindo uma classe de número de telefone vazia:

```
class TelephoneNumber {
}
```

Essa foi fácil! Em seguida, crio uma instância de número de telefone quando construo a pessoa:

class Person…

```
constructor() {
  this._telephoneNumber = new TelephoneNumber();
}
```

class TelephoneNumber…

```
get officeAreaCode() {return this._officeAreaCode;}
set officeAreaCode(arg) {this._officeAreaCode = arg;}
```

Em seguida, uso *Mover campo (Move Field)* em um dos campos.

class Person…

```
get officeAreaCode() {return this._telephoneNumber.officeAreaCode;}
set officeAreaCode(arg) {this._telephoneNumber.officeAreaCode = arg;}
```

Testo, e então movo o próximo campo.

class TelephoneNumber…

```
get officeNumber() {return this._officeNumber;}
set officeNumber(arg) {this._officeNumber = arg;}
```

class Person…

```
  get officeNumber() {return this._telephoneNumber.officeNumber;}
  set officeNumber(arg) {this._telephoneNumber.officeNumber = arg;}
```

Testo novamente e, em seguida, movo o método do número de telefone.

class TelephoneNumber…

```
  get telephoneNumber() {return `(${this.officeAreaCode}) ${this.officeNumber}`;}
```

class Person…

```
  get telephoneNumber() {return this._telephoneNumber.telephoneNumber;}
```

Agora tenho de organizar o código. Ter "office" como parte do código da classe de número de telefone não faz sentido, portanto eu renomeio.

class TelephoneNumber…

```
  get areaCode() {return this._areaCode;}
  set areaCode(arg) {this._areaCode = arg;}
  get number() {return this._number;}
  set number(arg) {this._number = arg;}
```

class Person…

```
  get officeAreaCode() {return this._telephoneNumber.areaCode;}
  set officeAreaCode(arg) {this._telephoneNumber.areaCode = arg;}
  get officeNumber() {return this._telephoneNumber.number;}
  set officeNumber(arg) {this._telephoneNumber.number = arg;}
```

O método com número de telefone no nome na classe de número de telefone também não faz muito sentido, portanto aplico *Renomear função (Rename Function)*.

class TelephoneNumber…

```
  toString() {return `(${this.areaCode}) ${this.number}`;}
```

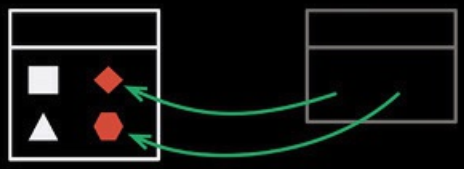
class Person…

```
  get telephoneNumber() {return this._telephoneNumber.toString();}
```

Números de telefone em geral são úteis, portanto acho que vou expor o novo objeto aos clientes. Posso substituir aqueles métodos "office" por métodos de acesso para o número de telefone. Dessa forma, porém, o número de telefone funcionará melhor como um Objeto de valor (Value Object) [mf-vo], portanto eu aplicaria *Mudar referência para valor (Change Reference to Value)* antes (o exemplo dessa refatoração mostra como eu faria isso para o número de telefone).

Internalizar classe (Inline Class)

inversa de: *Extrair classe (Extract Class)*



```
class Person {
  get officeAreaCode() {return
this._telephoneNumber.areaCode;}
  get officeNumber() {return this._telephoneNumber.number;}
}
class TelephoneNumber {
  get areaCode() {return this._areaCode;}
  get number() {return this._number;}
}
```

```
class Person {
  get officeAreaCode() {return
this._officeAreaCode;}
  get officeNumber() {return this._officeNumber;}
```

Motivação

Internalizar classe é o inverso de *Extrair classe (Extract Class)*. Uso *Internalizar classe* se constatar que não vale mais a pena tê-la, isto é, ela não deveria mais estar presente. Com frequência, isso é resultado de uma refatoração que move responsabilidades para fora da classe, restando pouca coisa nela. A essa altura, desdobro a classe em outra – em uma que faça mais uso da classe pequena.

Outro motivo para usar *Internalizar classe* é quando tenho duas classes que quero refatorar em um par de classes com uma alocação diferente de recursos. Posso achar mais fácil usar *Internalizar classe* antes de modo a combiná-las em uma única classe, e então usar *Extrair classe (Extract Class)* para fazer a nova separação. Esta é a abordagem genérica para reorganizar o código: às vezes, é mais fácil mover elementos, um de cada vez, de um

contexto para outro; outras vezes, porém, é melhor usar uma refatoração de internalização para reunir os contextos e depois usar uma refatoração de extração para separá-los em elementos diferentes.

Procedimento

- Na classe final, crie funções para todas as funções públicas da classe original. Essas funções devem simplesmente delegar para a classe original.
- Altere todas as referências aos métodos da classe original para que usem os métodos de delegação da classe final. Teste após cada mudança.
- Mova todas as funções e dados da classe original para a classe final, testando após cada mudança, até que a classe original esteja vazia.
- Apague a classe original a faça uma pequena cerimônia fúnebre simples.

Exemplo

Eis uma classe que armazena duas informações de monitoração de uma remessa.

```
class TrackingInformation {
  get shippingCompany() {return this._shippingCompany;}
  set shippingCompany(arg) {this._shippingCompany = arg;}
  get trackingNumber() {return this._trackingNumber;}
  set trackingNumber(arg) {this._trackingNumber = arg;}
  get display() {
    return `${this.shippingCompany}: ${this.trackingNumber}`;
  }
}
```

Ela é usada como parte de uma classe de remessa (shipment).

class Shipment…

```
get trackingInfo() {
  return this._trackingInformation.display;
}
get trackingInformation() {return this._trackingInformation;}
set trackingInformation(aTrackingInformation) {
  this._trackingInformation = aTrackingInformation;
}
```

Embora tenha valido a pena ter essa classe no passado, não acho mais que vale a pena tê-la, portanto quero internalizá-la em Shipment.

Começo olhando os lugares que chamam os métodos de TrackingInformation.

chamada...

```
aShipment.trackingInformation.shippingCompany = request.vendor;
```

Passarei todas as funções desse tipo para Shipment, mas faço isso de modo um pouco diferente de como geralmente faço um *Mover função (Move Function)*. Nesse caso, começo colocando um método de delegação na classe de remessa e adaptando o cliente para chamá-lo.

class Shipment…

```
set shippingCompany(arg) {this._trackingInformation.shippingCompany = arg;}
```

chamada...

```
aShipment.~~trackingInformation.~~shippingCompany = request.vendor;
```

Faço isso para todos os elementos das informações de acompanhamento da remessa usados pelos clientes. Feito isso, posso mover todos os elementos dessa classe para a classe de remessa.

Começo aplicando *Internalizar função (Inline Function)* no método display.

class Shipment…

```
get trackingInfo() {
  return `${this.shippingCompany}: ${this.trackingNumber}`;
}
```

Movo o campo referente à empresa de remessa (shipping company).

```
get shippingCompany() {return this.~~_trackingInformation.~~_shippingCompany;}
set shippingCompany(arg) {this.~~_trackingInformation.~~_shippingCompany = arg;}
```

Não uso o procedimento completo de *Mover campo (Move Field)*, pois, nesse caso, só referencio shippingCompany a partir de Shipment, que é o alvo da mudança. Desse modo, não preciso dos passos que colocam uma referência da origem para o destino.

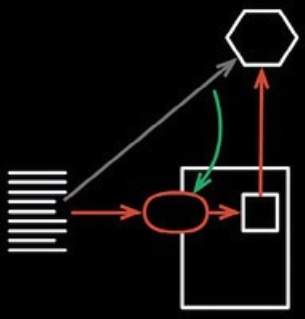
Continuo até que tudo tenha sido transferido. Depois que fizer isso, posso apagar a classe de informações de acompanhamento da remessa.

class Shipment…

```
get trackingInfo() {
  return `${this.shippingCompany}: ${this.trackingNumber}`;
}
get shippingCompany() {return this._shippingCompany;}
set shippingCompany(arg) {this._shippingCompany = arg;}
get trackingNumber() {return this._trackingNumber;}
set trackingNumber(arg) {this._trackingNumber = arg;}
```

Ocultar delegação (Hide Delegate)

inversa de: *Remover intermediário (Remove Middle Man)*



```
manager =
aPerson.department.manager;
```

```
manager = aPerson.manager;

class Person {
  get manager() {return
this.department.manager;}
```

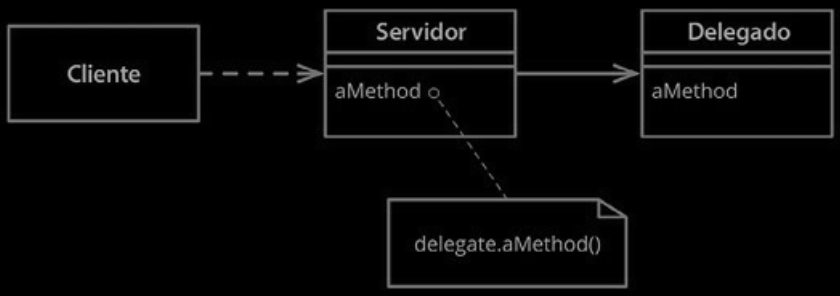
Motivação

Um dos segredos – se não *o* segredo – de um bom design modular é o encapsulamento. Um encapsulamento significa que os módulos precisam saber menos sobre outras partes do sistema. Então, quando houver mudanças, menos módulos precisarão ter conhecimento delas – facilitando fazer a mudança.

Quando começamos a aprender sobre orientação a objetos, nos ensinam que encapsulamento significa ocultar nossos campos. À medida que nos tornamos mais sofisticados, percebemos que há outros itens que podemos encapsular.

Se eu tiver um código de cliente que chame um método definido em um objeto que está um campo de um objeto servidor, o cliente deverá conhecer esse objeto delegado. Se esse objeto mudar a sua interface, as mudanças se propagarão para todos os clientes do servidor que usam o objeto delegado. Posso eliminar essa dependência colocando um método simples de delegação no servidor que oculte o objeto delegado. Então, qualquer modificação que eu fizer no objeto delegado se propagará somente até o servidor, e não aos

clientes.



Procedimento

- Para cada método no objeto delegado, crie um método de delegação simples no servidor.
- Adapte o cliente para que chame o servidor. Teste após cada mudança.
- Se não houver mais clientes precisando acessar o objeto delegado, remova o método de acesso do servidor para o objeto delegado.
- Teste.

Exemplo

Começarei com uma pessoa e um departamento.

class Person…

```
constructor(name) {
  this._name = name;
}
get name() {return this._name;}
get department() {return this._department;}
set department(arg) {this._department = arg;}
```

class Department…

```
get chargeCode() {return this._chargeCode;}
set chargeCode(arg) {this._chargeCode = arg;}
get manager() {return this._manager;}
set manager(arg) {this._manager = arg;}
```

Um código de cliente quer saber quem é o gerente de uma pessoa. Para isso, é necessário obter antes a informação sobre o departamento.

código do cliente…

```
manager = aPerson.department.manager;
```

Isso revela ao cliente como a classe departamento funciona e o fato de que ela é responsável por monitorar a informação sobre o gerente. Posso reduzir esse acoplamento ocultando a classe departamento do cliente. Faço isso criando um método simples de delegação na classe pessoa:

class Person…

```
get manager() {return this._department.manager;}
```

Preciso agora alterar todos os clientes da classe pessoa para que usem esse novo método.

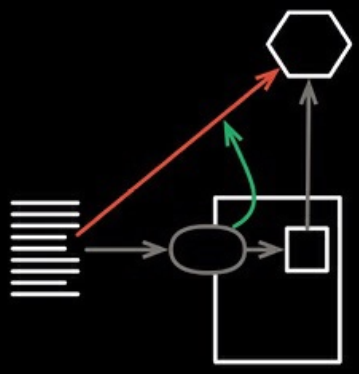
código do cliente…

```
manager = aPerson.department.manager;
```

Depois de ter feito a alteração para todos os métodos da classe departamento e todos os clientes da classe pessoa, posso remover o método de acesso a department na classe pessoa.

Remover intermediário (Remove Middle Man)

inversa de: *Ocultar delegação (Hide Delegate)*



```
manager = aPerson.manager;

class Person {
  get manager() {return
this.department.manager;}
```

```
manager =
aPerson.department.manager;
```

Motivação

Na motivação de *Ocultar delegação (Hide Delegate)*, falei das vantagens de encapsular o uso de um objeto delegado. Há um preço para isso. Sempre que o cliente quiser usar um novo recurso do objeto delegado, tenho de adicionar um método simples de delegação no servidor. Depois de adicionar recursos por um tempo, fico irritado com todo esse encaminhamento. A classe de servidor é apenas um intermediário (middle man) (ver a seção *Intermediário* do Capítulo 3 na *página* 106), e talvez seja hora de o cliente chamar diretamente o objeto delegado. (Esse cheiro com frequência aparece quando as pessoas se entusiasmam demais em seguir a Lei de Demeter, que eu acharia muito melhor se fosse chamada de Sugestão de Demeter Ocasionalmente Útil.)

É difícil determinar o nível correto para ocultar itens. Felizmente, com *Ocultar delegação (Hide Delegate)* e *Remover intermediário (Remove Middle Man)*, isso não importa tanto. Posso adaptar meu código com o passar do tempo. À medida que o sistema mudar, o critério para determinar quanto devo ocultar também mudará. Um bom encapsulamento seis meses atrás poderá ser inconveniente agora. Refatoração significa jamais ter de lamentar – eu apenas faço a correção.

Procedimento

- Crie um getter para o objeto delegado.
- Para cada uso de um método de delegação por um cliente, substitua a chamada ao método de delegação fazendo o encadeamento com o método de acesso. Teste após cada substituição.

Se todas as chamadas a um método de delegação forem substituídas, você poderá apagar o método de delegação.

Com refatorações automatizadas, você pode usar *Encapsular variável (Encapsulate Variable)* no campo do objeto delegado e então *Internalizar função (Inline Function)* em todos os métodos que o usam.

Exemplo

Começarei com uma classe pessoa que usa um objeto departamento associado para determinar um gerente. (Se você estiver lendo este livro sequencialmente, esse exemplo talvez lhe seja estranhamente familiar.)

código do cliente…

```
manager = aPerson.manager;
```

class Person…

```
get manager() {return this._department.manager;}
```

class Department…

```
get manager() {return this._manager;}
```

Esse código serve apenas para usar e encapsular o departamento. No entanto, se muitos métodos estiverem fazendo isso, acabarei com várias dessas delegações simples na classe pessoa. Nesse caso, será interessante remover o intermediário (middle man). Em primeiro lugar, crio um método de acesso para o objeto delegado:

class Person…

```
get department() {return this._department;}
```

Agora, modifico um cliente de cada vez para que usem diretamente o departamento.

código do cliente…

```
manager = aPerson.department.manager;
```

Depois que fizer isso com todos os clientes, posso remover o método de Person para obter o gerente. É possível repetir esse processo para quaisquer outras delegações simples em Person.

É possível fazer uma mistura nesse caso. Algumas delegações podem ser tão comuns que eu preferiria mantê-las para que seja mais fácil trabalhar com o código do cliente. Não há absolutamente nenhum motivo para decidir se devo ocultar um objeto delegado ou se devo remover um intermediário – circunstâncias específicas sugerem qual abordagem deve ser adotada, e pessoas sensatas podem divergir no que diz respeito à abordagem que funcione melhor.

Se eu tiver refatorações automatizadas, haverá uma boa variação nesses passos. Em primeiro lugar, uso *Encapsular variável (Encapsulate Variable)* em department. Com isso, em vez de usar o getter do gerente, o getter público de departamento é utilizado:

class Person…

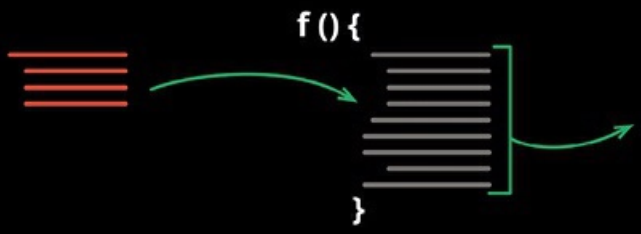
```
get manager() {return this.department.manager;}
```

A mudança é muito sutil em JavaScript, mas, ao remover o underscore de

department, *estou usando o novo getter em vez de acessar o campo diretamente.*

Então aplico *Internalizar função (Inline Function)* no método do gerente para substituir todas as chamadas de uma só vez.

Substituir algoritmo (Substitute Algorithm)




```
function foundPerson(people) {
  for(let i = 0; i < people.length;
i++) {
    if (people[i] === "Don") {
      return "Don";
    }
    if (people[i] === "John") {
      return "John";
    }
    if (people[i] === "Kent") {
      return "Kent";
    }
  }
  return "";
}
```

```
function foundPerson(people) {
  const candidates = ["Don", "John", "Kent"];
  return people.find(p => candidates.includes(p))
|| '';
}
```

Motivação

Nunca tentei escalpelar um gato. Já me disseram que há várias maneiras de

fazer isso. Tenho certeza de que algumas são mais fáceis que outras. O mesmo vale para os algoritmos. Se eu achar uma maneira mais clara de fazer algo, substituirei o modo complicado pelo modo mais claro. A refatoração pode separar algo complexo em partes mais simples; às vezes, porém, atinjo o ponto em que tenho de remover o algoritmo completo e substituí-lo por algo mais simples. Isso ocorre à medida que aprendo mais sobre o problema e percebo que há um modo mais fácil de fazer uma tarefa. Também ocorre quando começo a usar uma biblioteca que ofereça recursos que dupliquem meu código.

Às vezes, quando quero modificar o algoritmo para que funcione de modo um pouco diferente, é mais fácil começar substituindo-o por algo que deixe minha modificação mais simples de ser feita.

Quando for necessário executar esse passo, tenho de me certificar de que decompus o método o máximo possível. Substituir um algoritmo grande e complexo é muito difícil: somente o deixando mais simples posso deixar a substituição mais controlada.

Procedimento

- Organize o código a ser substituído de modo que ele preencha uma função completa.
- Prepare testes usando somente essa função para capturar seu comportamento.
- Prepare seu algoritmo alternativo.
- Execute verificações estáticas.
- Execute testes para comparar a saída do algoritmo antigo com a saída do algoritmo novo. Se forem iguais, seu trabalho estará terminado. Caso contrário, use o algoritmo antigo para comparação nos testes e na depuração.

CAPÍTULO 8

Movendo recursos

Até agora, as refatorações estavam relacionadas a criar, remover e renomear elementos de programa. Outra parte importante da refatoração é mover elementos entre contextos. Uso *Mover função (Move Function)* para mover funções entre classes e outros módulos. Campos também podem ser movidos com *Mover campo (Move Field)*.

Posso igualmente mover instruções individuais. Uso *Mover instruções para uma função (Move Statements into Function)* e *Mover instruções para os pontos de chamada (Move Statements to Callers)* para movê-las para dentro ou para fora de funções, assim como *Deslocar instruções (Slide Statements)* para movê-las no interior de uma função. Às vezes, posso tomar algumas instruções que correspondam a uma função existente e usar *Substituir código internalizado por chamada de função (Replace Inline Code with Function Call)* para remover a duplicação.

Duas refatorações que uso frequentemente com laços são *Dividir laço (Split Loop)*, para garantir que um laço faça apenas uma tarefa, e *Substituir laço por pipeline (Replace Loop with Pipeline)*, para me livrar totalmente de um laço.

Por fim, temos a refatoração preferida de muitos bons programadores: *Remover código morto (Remove Dead Code)*. Não há nada tão satisfatório quanto aplicar o lança-chamas digital em instruções supérfluas.

Mover função (Move Function)

anteriormente: *Mover método* (Move Method)

```
class Account {
```

```
    get overdraftCharge()
{...}
```

```
class AccountType {
    get overdraftCharge()
{...}
```

Motivação

O coração de um bom design de software é a sua modularidade – que é minha capacidade de fazer a maioria das modificações em um programa tendo de compreender somente uma pequena parte dele. Para alcançar essa modularidade, é necessário garantir que elementos de software relacionados estejam agrupados e as ligações entre eles sejam fáceis de encontrar e de entender. No entanto, minha compreensão de como fazer isso não é estática – à medida que entendo melhor o que estou fazendo, aprendo a agrupar melhor os elementos de software. Para que essa compreensão crescente se reflita no código, devo mover os elementos.

Todas as funções estão em algum contexto; este pode ser global, mas, em geral, é alguma espécie de módulo. Em um programa orientado a objetos, o contexto principal dos módulos é uma classe. Aninhar uma função em outra cria outro contexto comum. Diferentes linguagens oferecem formas variadas de modularidade, cada qual criando um contexto para uma função ser inserida.

Um dos motivos mais óbvios para mover uma função é quando ela referencia mais elementos de outros contextos do que do contexto em que se encontra no momento. Movê-la para junto desses elementos com frequência melhora o encapsulamento, permitindo que outras partes do software sejam menos dependentes dos detalhes desse módulo.

De modo semelhante, posso mover uma função por causa do local em que estão suas chamadas, ou de onde devo chamá-la em minha próxima melhoria do código. Uma função definida como auxiliar dentro de outra função pode ter valor por si só, portanto valerá a pena movê-la para outro lugar mais acessível. Um método de uma classe talvez seja mais facilmente usado se for movido para outra classe.

A decisão de mover uma função raramente é fácil. Para me ajudar a decidir,

analiso o contexto atual e o contexto candidato para receber essa função. É necessário observar as funções que chamam essa função, as que são chamadas pela função sendo movida e quais dados ela utiliza. Muitas vezes, percebo que preciso de um novo contexto para um grupo de funções, e crio um usando *Combinar funções em classe (Combine Functions into Class)* ou *Extrair classe (Extract Class)*. Embora seja difícil decidir qual é o melhor lugar para uma função, quanto mais difícil for essa escolha, em geral menos importante ela será. Acho relevante tentar trabalhar com funções em um contexto, sabendo que aprenderei até que ponto elas serão apropriadas a esse contexto; caso não sejam, é sempre possível movê-las depois.

Procedimento

- Analise todos os elementos de programa usados pela função escolhida em seu contexto atual. Considere se eles devem ser movidos também.

Se eu identificar uma função chamada que também deva ser movida, geralmente a movo antes. Dessa forma, mover um grupo de funções começa por aquela que apresentar a menor dependência em relação às outras funções do grupo.

Se uma função de alto nível for a única a chamar as subfunções, você poderá internalizar essas funções no método de alto nível, movê-las e fazer uma extração novamente no destino.

- Verifique se a função escolhida é um método polimórfico.

Se eu estiver usando uma linguagem orientada a objetos, tenho de levar em consideração as declarações de superclasse e subclasse.

- Copie a função para o contexto final. Faça as adaptações para que ela se adeque ao seu novo lar.

Se o corpo da função usar elementos do contexto original, é necessário passá-los como parâmetros ou passar uma referência a esse contexto.

Mover uma função com frequência significa que é necessário criar um nome diferente, mais apropriado ao novo contexto.

- Faça análises estáticas.
- Descubra como referenciar a função final a partir do contexto original.
- Transforme a função original em uma função de delegação.

- Teste.
- Considere usar *Internalizar função (Inline Function)* na função original.

A função original pode permanecer indefinidamente como uma função de delegação. Contudo, se quem a chama puder acessar facilmente a função final de forma direta, será melhor remover o intermediário.

Exemplo: movendo uma função aninhada para o nível mais alto

Começarei com uma função que calcula a distância total para um registro de rota de GPS.

```
function trackSummary(points) {
  const totalTime = calculateTime();
  const totalDistance = calculateDistance();
  const pace = totalTime / 60 / totalDistance ;
  return {
    time: totalTime,
    distance: totalDistance,
    pace: pace
  };

  function calculateDistance() {
    let result = 0;
    for (let i = 1; i < points.length; i++) {
      result += distance(points[i-1], points[i]);
    }
    return result;
  }
  function distance(p1,p2) { ... }
  function radians(degrees) { ... }
  function calculateTime() { ... }
}
```

Gostaria de mover `calculateDistance` para o nível mais alto a fim de que eu calcule distâncias de rotas sem todas as demais partes do resumo da rota.

Começo copiando a função para o nível mais alto.

```
function trackSummary(points) {
  const totalTime = calculateTime();
  const totalDistance = calculateDistance();
  const pace = totalTime / 60 / totalDistance ;
  return {
```

```
    time: totalTime,
    distance: totalDistance,
    pace: pace
  };

  function calculateDistance() {
    let result = 0;
    for (let i = 1; i < points.length; i++) {
      result += distance(points[i-1], points[i]);
    }
    return result;
  }
  ...
  function distance(p1,p2) { ... }
  function radians(degrees) { ... }
  function calculateTime() { ... }
}
function top_calculateDistance() {
  let result = 0;
  for (let i = 1; i < points.length; i++) {
    result += distance(points[i-1], points[i]);
  }
  return result;
}
```

Quando copio uma função como essa, gosto de alterar seu nome para que eu possa distingui-la tanto no código quanto na minha cabeça. Não quero pensar em qual deveria ser o nome correto nesse momento, portanto crio um nome temporário.

O programa continua funcionando, mas minha análise estática se mostra insatisfeita, com razão. A nova função tem dois símbolos indefinidos: `distance` e `points`. O modo natural de lidar com `points` é passá-lo como parâmetro.

```
function top_calculateDistance(points) {
  let result = 0;
  for (let i = 1; i < points.length; i++) {
    result += distance(points[i-1], points[i]);
  }
  return result;
}
```

Eu poderia fazer o mesmo com `distance`, mas talvez faça mais sentido movê-lo junto com `calculateDistance`. Eis o código relevante:

function trackSummary…

```
function distance(p1,p2) {
  // fórmula de Haversine: veja http://www.movable-type.co.uk/scripts/latlong.html
  const EARTH_RADIUS = 3959; // em milhas
  const dLat = radians(p2.lat) - radians(p1.lat);
  const dLon = radians(p2.lon) - radians(p1.lon);
  const a = Math.pow(Math.sin(dLat / 2),2)
        + Math.cos(radians(p2.lat))
        * Math.cos(radians(p1.lat))
        * Math.pow(Math.sin(dLon / 2), 2);
  const c = 2 * Math.atan2(Math.sqrt(a), Math.sqrt(1-a));
  return EARTH_RADIUS * c;
}
function radians(degrees) {
  return degrees * Math.PI / 180;
}
```

Posso ver que `distance` usa apenas `radians`, e `radians` não usa nada além de seu contexto atual. Então, em vez de passar as funções, é possível movê-las também. Posso dar um pequeno passo nessa direção movendo-as de seu contexto atual para deixá-las aninhadas na função `calculateDistance` aninhada.

```
function trackSummary(points) {
  const totalTime = calculateTime();
  const totalDistance = calculateDistance();
  const pace = totalTime / 60 / totalDistance ;
  return {
    time: totalTime,
    distance: totalDistance,
    pace: pace
  };

  function calculateDistance() {
    let result = 0;
    for (let i = 1; i < points.length; i++) {
      result += distance(points[i-1], points[i]);
    }
    return result;
    function distance(p1,p2) { ... }
    function radians(degrees) { ... }
  }
```

Ao fazer isso, posso usar tanto a análise estática quando os testes para me dizer se há alguma complicação. Nesse caso, tudo está bem, portanto posso copiá-las para `top_calculateDistance`.

```
function top_calculateDistance(points) {
```

```
    let result = 0;
    for (let i = 1; i < points.length; i++) {
      result += distance(points[i-1], points[i]);
    }
    return result;
    function distance(p1,p2) { ... }
    function radians(degrees) { ... }
  }
```

Novamente, a cópia não altera o modo como o programa executa, mas me dá uma oportunidade para fazer mais análises estáticas. Se eu não tivesse percebido que `distance` chama `radians`, o linter teria identificado esse problema nesse passo.

Agora que já preparei o cenário, é hora da alteração principal – o corpo do `calculateDistance` original agora chamará `top_calculateDistance`:

```
function trackSummary(points) {
  const totalTime = calculateTime();
  const totalDistance = calculateDistance();
  const pace = totalTime / 60 / totalDistance ;
  return {
    time: totalTime,
    distance: totalDistance,
    pace: pace
  };

  function calculateDistance() {
    return top_calculateDistance(points);
  }
```

Esse é o momento crucial para executar testes, a fim de verificar, de modo completo, se a função movida está bem acomodada em seu novo lar.

Depois de fazer isso, é como abrir as caixas após uma mudança de casa. A primeira tarefa é decidir se a função original, que está somente fazendo uma delegação, deve ser mantida ou não. Nesse caso, há poucas chamadas, e, como ocorre em geral com funções aninhadas, elas são extremamente localizadas. Desse modo, fico feliz em me livrar dela.

```
function trackSummary(points) {
  const totalTime = calculateTime();
  const totalDistance = top_calculateDistance(points);
  const pace = totalTime / 60 / totalDistance ;
  return {
    time: totalTime,
```

```
    distance: totalDistance,
    pace: pace
  };
```

Agora também é uma boa hora para pensar no nome que quero dar à função. Como a função de nível mais alto tem mais visibilidade, gostaria que ela tivesse o melhor nome possível. totalDistance parece uma boa escolha. Não posso usar esse nome imediatamente, pois ele será encoberto pela variável em trackSummary – de qualquer maneira, não vejo motivo algum para manter a variável, portanto uso *Internalizar variável (Inline Variable)*.

```
function trackSummary(points) {
  const totalTime = calculateTime();
  const pace = totalTime / 60 / totalDistance(points) ;
  return {
    time: totalTime,
    distance: totalDistance(points),
    pace: pace
  };
function totalDistance(points) {
  let result = 0;
  for (let i = 1; i < points.length; i++) {
    result += distance(points[i-1], points[i]);
  }
  return result;
```

Se houvesse a necessidade de manter a variável, eu a teria renomeado com algo como totalDistanceCache ou distance.

Como as funções para distance e radians não dependem de nada que está em totalDistance, prefiro movê-las para o nível mais alto também, colocando todas as quatro funções nesse nível.

```
function trackSummary(points) { ... }
function totalDistance(points) { ... }
function distance(p1,p2) { ... }
function radians(degrees) { ... }
```

Algumas pessoas iriam preferir manter distance e radians em totalDistance a fim de restringir sua visibilidade. Em algumas linguagens, isso pode ser considerado, mas, na ES 2015, o JavaScript tem um sistema excelente de módulos, que é a melhor ferramenta para controlar a visibilidade de funções. Em geral, sou cauteloso com funções aninhadas – elas estabelecem facilmente inter-relações ocultas entre os dados, as quais podem ser difíceis

de entender.

Exemplo: movendo entre classes

Para ilustrar essa variante de *Mover função*, começarei com:

class Account…

```
get bankCharge() {
  let result = 4.5;
  if (this._daysOverdrawn > 0) result += this.overdraftCharge;
  return result;
}
get overdraftCharge() {
  if (this.type.isPremium) {
    const baseCharge = 10;
    if (this.daysOverdrawn <= 7)
      return baseCharge;
    else
      return baseCharge + (this.daysOverdrawn - 7) * 0.85;
  }
  else
    return this.daysOverdrawn * 1.75;
}
```

A seguir, teremos mudanças que resultarão em diferentes tipos de conta com diferentes algoritmos para determinar a cobrança. Assim, parece natural mover `overdraftCharge` para a classe do tipo de conta.

O primeiro passo é observar os recursos que o método `overdraftCharge` usa e considerar se vale a pena mover um conjunto de métodos. Nesse caso, é necessário que o método `daysOverdrawn` permaneça na classe de conta, pois variará de acordo com as contas individuais.

Em seguida, copio o corpo do método para a classe de tipo da conta (account type) e faço as adaptações em seu código.

class AccountType…

```
overdraftCharge(daysOverdrawn) {
  if (this.isPremium) {
    const baseCharge = 10;
    if (daysOverdrawn <= 7)
      return baseCharge;
    else
      return baseCharge + (daysOverdrawn - 7) * 0.85;
  }
```

```
    else
      return daysOverdrawn * 1.75;
  }
```

Para que o método esteja adequado ao seu novo local, tenho de lidar com duas chamadas que tiveram seu escopo alterado. isPremium agora é uma chamada simples em this. Com daysOverdrawn, devo decidir se passo o valor ou a conta. Por enquanto, passarei apenas o valor simples, mas posso muito bem mudar de ideia no futuro caso precise de outras informações além dos dias em que o cheque especial foi usado na conta (days overdrawn) – especialmente se a informação que eu quiser da conta variar de acordo com o tipo da conta.

Em seguida, substituo o corpo do método original por uma chamada de delegação.

class Account…

```
  get bankCharge() {
    let result = 4.5;
    if (this._daysOverdrawn > 0) result += this.overdraftCharge;
    return result;
  }

  get overdraftCharge() {
    return this.type.overdraftCharge(this.daysOverdrawn);
  }
```

Na sequência, devo decidir se deixo a delegação ou se internalizo overdraftCharge. A internalização resulta no seguinte código:

class Account…

```
  get bankCharge() {
    let result = 4.5;
    if (this._daysOverdrawn > 0)
      result += this.type.overdraftCharge(this.daysOverdrawn);
    return result;
  }
```

Nos passos anteriores, passei daysOverdrawn como parâmetro – mas, se houver muitos dados da conta para serem passados, seria preferível passar a própria conta.

class Account…

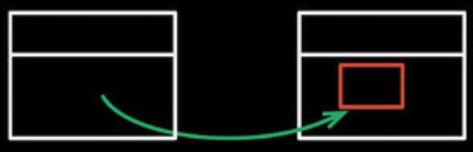
```
  get bankCharge() {
```

```
    let result = 4.5;
    if (this._daysOverdrawn > 0) result += this.overdraftCharge;
    return result;
  }

  get overdraftCharge() {
    return this.type.overdraftCharge(this);
  }
```

class AccountType…

```
  overdraftCharge(account) {
    if (this.isPremium) {
      const baseCharge = 10;
      if (account.daysOverdrawn <= 7)
        return baseCharge;
      else
        return baseCharge + (account.daysOverdrawn - 7) * 0.85;
    }
    else
      return account.daysOverdrawn * 1.75;
  }
```

Mover campo (Move Field)



```
class Customer {
  get plan() {return this._plan;}
  get discountRate() {return
this._discountRate;}
```

```
class Customer {
  get plan() {return this._plan;}
  get discountRate() {return
this.plan.discountRate;}
```

Motivação

A programação envolve escrever muito código para implementar comportamentos – mas a força de um programa na verdade se encontra em

suas estruturas de dados. Se eu tiver um bom conjunto de estruturas de dados apropriadas ao problema, meu código que implementa os comportamentos será simples e organizado. Contudo, estruturas de dados ruins resultam em muito código cuja tarefa é apenas lidar com dados ruins. E não é somente um código confuso que é mais difícil de entender; significa também que as estruturas de dados encobrirão o que o programa faz.

Desse modo, as estruturas de dados são importantes – mas, como na maioria dos casos em programação, é difícil criá-las corretamente. Faço uma análise inicial a fim de determinar as melhores estruturas de dados possíveis, e percebo que experiência e técnicas como design orientado a domínios têm melhorado minha habilidade para fazer isso. Todavia, apesar de todas as minhas habilidades e experiência, ainda noto que, com frequência, cometo erros nesse design inicial. Durante o processo da programação, aprendo mais sobre o domínio do problema e minhas estruturas de dados. Uma decisão de design que seja razoável e correta em uma semana poderá se tornar incorreta em outra.

Assim que percebo que uma estrutura de dados não está correta, é essencial alterá-la. Se eu deixar minhas estruturas de dados com defeitos, eles confundirão meu raciocínio e complicarão meu código muito além no futuro.

Posso procurar mover dados ao perceber que tenho sempre de passar um campo de um registro quando passo outro registro para uma função. Seria melhor que porções de dados que são sempre passadas em conjunto estivessem em um único registro para que seu relacionamento esteja claro. A mudança também é um fator a ser considerado: se uma mudança em um registro fizer com que um campo de outro registro também mude, é sinal de que há um campo no lugar errado. Se eu tiver de atualizar o mesmo campo em várias estruturas, é sinal de que esse campo deve ser movido para outro lugar, de modo que precise ser atualizado somente uma vez.

Em geral, executo *Mover campo* no contexto de um conjunto mais amplo de mudanças. Depois que movo um campo, percebo que, para muitos usuários desse campo, seria melhor acessar os dados por meio do objeto final em vez de usar o código original. Então os altero depois, com refatorações. De modo semelhante, talvez eu descubra que não é possível executar *Mover campo* no momento em virtude do modo como os dados estão sendo usados. É necessário refatorar alguns padrões de uso antes, e então mover o código.

Em minha descrição até agora, menciono “registro”, mas tudo isso é válido para classes e objetos também. Uma classe é um tipo de registro com funções associadas – e elas precisam se manter saudáveis tanto quanto qualquer outro dado. As funções associadas facilitam mover os dados, pois eles estão encapsulados por métodos de acesso. Posso mover os dados, modificar os métodos de acesso, e os clientes desses métodos continuarão funcionando. Assim, essa é uma refatoração mais fácil de fazer se você tiver classes, e minha descrição a seguir parte desse pressuposto. Se eu estiver usando registros comuns, sem suporte para encapsulamento, ainda é possível fazer uma modificação como essa, porém será mais complicado.

Procedimento

- Garanta que o campo original esteja encapsulado.
- Teste.
- Crie um campo (e métodos de acesso) no código final.
- Execute verificações estáticas.
- Garanta que haja uma referência do objeto original para o objeto final.

Um campo ou método existente pode lhe fornecer acesso ao objeto final. Caso contrário, veja se você consegue facilmente criar um método que faça isso. Se falhar, talvez seja necessário criar um novo campo no objeto original, capaz de armazenar o objeto final. Essa pode ser uma alteração permanente, mas também pode ser temporária, até que você tenha feito refatorações suficientes no contexto mais amplo.

- Adapte os métodos de acesso para que usem o campo final.

Se o campo final for compartilhado entre os objetos originais, considere antes atualizar o setter para modificar tanto o campo final quanto o campo original, seguido de *Introduzir asserção (Introduce Assertion)* para detectar atualizações inconsistentes. Depois que determinar que tudo está bem, finalize as mudanças nos métodos de acesso para que usem o campo final.

- Teste.
- Remova o campo original.
- Teste.

Exemplo

Começarei este exemplo com o cliente e o contrato a seguir:

class Customer…

```
constructor(name, discountRate) {
  this._name = name;
  this._discountRate = discountRate;
  this._contract = new CustomerContract(dateToday());
}
get discountRate() {return this._discountRate;}
becomePreferred() {
  this._discountRate += 0.03;
  // outras coisas interessantes
}
applyDiscount(amount) {
  return amount.subtract(amount.multiply(this._discountRate));
}
```

class CustomerContract…

```
  constructor(startDate) {
    this._startDate = startDate;
  }
```

Quero mover o campo da taxa de desconto (discount rate) do cliente (customer) para o contrato do cliente (customer contract).

A primeira tarefa a ser feita é usar *Encapsular variável (Encapsulate Variable)* para encapsular o acesso ao campo da taxa de desconto.

class Customer…

```
constructor(name, discountRate) {
  this._name = name;
  this._setDiscountRate(discountRate);
  this._contract = new CustomerContract(dateToday());
}
get discountRate() {return this._discountRate;}
_setDiscountRate(aNumber) {this._discountRate = aNumber;}
becomePreferred() {
  this._setDiscountRate(this.discountRate + 0.03);
  // outras coisas interessantes
}
applyDiscount(amount) {
  return amount.subtract(amount.multiply(this.discountRate));
}
```

Uso um método para atualizar a taxa de desconto em vez de um setter de propriedade, pois não quero criar um setter público para a taxa.

Adiciono um campo e métodos de acesso na classe de contrato do cliente.

class CustomerContract…

```
constructor(startDate, discountRate) {
  this._startDate = startDate;
  this._discountRate = discountRate;
}
get discountRate() {return this._discountRate;}
set discountRate(arg) {this._discountRate = arg;}
```

Agora modifico os métodos de acesso no cliente para que usem o novo campo. Quando fiz isso, obtive um erro: “Cannot set property ‘discountRate’ of undefined” (Não é possível definir a propriedade ‘discountRate’ de indefinido). Isso ocorreu porque `_setDiscountRate` foi chamado antes que eu tivesse criado o objeto de contrato no construtor. Para corrigir isso, retornei antes ao estado anterior e então usei *Deslocar instruções (Slide Statements)* para mover `_setDiscountRate` para depois da criação do contrato.

class Customer…

```
constructor(name, discountRate) {
  this._name = name;
  this._setDiscountRate(discountRate);
  this._contract = new CustomerContract(dateToday());
}
```

Testei, e então alterei novamente os métodos de acesso para que usem o contrato.

class Customer…

```
get discountRate() {return this._contract.discountRate;}
_setDiscountRate(aNumber) {this._contract.discountRate = aNumber;}
```

Como estou usando JavaScript, não há nenhum campo original declarado, portanto não preciso remover mais nada.

Modificando um registro simples

Essa refatoração em geral é mais simples com objetos, pois o encapsulamento oferece um modo natural de encapsular o acesso aos dados com métodos. Se eu tiver muitas funções acessando um registro simples, então, embora a refatoração continue sendo importante, certamente ela será mais complicada.

Posso criar funções de acesso e modificar todas as leituras e escritas para que usem essas funções. Se o campo sendo movido for imutável, posso atualizar tanto o campo original quanto o campo final quando definir seu valor e migrar gradualmente as leituras. Apesar disso, se for possível, meu primeiro passo seria usar *Encapsular registro (Encapsulate Record)* para transformar o registro em uma classe, de modo que seja possível fazer a alteração mais facilmente.

Exemplo: movendo para um objeto compartilhado

Vamos agora considerar um caso diferente. Eis uma conta que tem uma taxa de juros (interest rate):

class Account…

```
constructor(number, type, interestRate) {
  this._number = number;
  this._type = type;
  this._interestRate = interestRate;
}
get interestRate() {return this._interestRate;}
```

class AccountType…

```
constructor(nameString) {
  this._name = nameString;
}
```

Quero alterar o código de modo que a taxa de juros de uma conta seja determinada pelo tipo da conta.

O acesso à taxa de juros já está devidamente encapsulado, portanto criarei apenas o campo e um método de acesso apropriado no tipo da conta.

class AccountType…

```
constructor(nameString, interestRate) {
  this._name = nameString;
  this._interestRate = interestRate;
}
get interestRate() {return this._interestRate;}
```

Contudo, há um problema em potencial quando atualizo os acessos a partir da conta. Antes dessa refatoração, cada conta tinha a própria taxa de juros. Agora, quero que todas as contas compartilhem as taxas de juros de seu tipo

de conta. Se todas as contas do mesmo tipo já tiverem a mesma taxa de juros, não haverá nenhuma mudança no comportamento observável, portanto não haverá problemas com a refatoração. Porém, se houver uma conta com uma taxa de juros diferente, não será mais uma refatoração. Se os dados de minha conta estiverem em um banco de dados, devo verificá-lo a fim de garantir que todas as contas tenham a taxa correspondente ao seu tipo. Também posso usar *Introduzir asserção (Introduce Assertion)* na classe de conta.

class Account…

```
constructor(number, type, interestRate) {
  this._number = number;
  this._type = type;
  assert(interestRate === this._type.interestRate);
  this._interestRate = interestRate;
}
get interestRate() {return this._interestRate;}
```

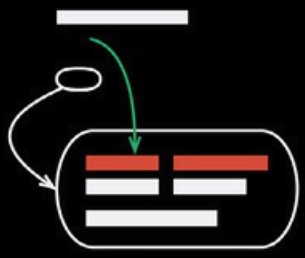
Posso executar o sistema por um tempo com essa asserção definida para ver se obtenho um erro. Ou, em vez de adicionar uma asserção, posso fazer log do problema. Depois que eu tiver certeza de que não estarei introduzindo uma mudança observável, posso alterar o acesso, removendo totalmente a atualização na conta.

class Account…

```
constructor(number, type) {
  this._number = number;
  this._type = type;
}
get interestRate() {return this._type.interestRate;}
```

Mover instruções para uma função (Move Statements into Function)

inversa de: *Mover instruções para os pontos de chamada (Move Statements to Callers)*



```
result.push(`<p>title: ${person.photo.title}
</p>`);
result.concat(photoData(person.photo));

function photoData(aPhoto) {
  return [
    `<p>location: ${aPhoto.location}</p>`,
    `<p>date: ${aPhoto.date.toDateString()}
</p>`,
  ];
}
```

```
result.concat(photoData(person.photo));

function photoData(aPhoto) {
  return [
    `<p>title: ${aPhoto.title}</p>`,
    `<p>location: ${aPhoto.location}</p>`,
    `<p>date: ${aPhoto.date.toDateString()}
</p>`,
  ];
}
```

Motivação

Remover duplicação é uma das melhores regras gerais de um código saudável. Se eu vir o mesmo código executado sempre que chamar uma função em particular, procurarei combinar esse código repetido na própria função. Dessa forma, quaisquer modificações futuras no código repetido poderão ser feitas em um só lugar e usadas por todos que o chamam. Caso o código precise variar no futuro, posso facilmente o mover (ou parte dele) de novo usando *Mover instruções para os pontos de chamada (Move Statements to Callers)*.

Movo instruções para uma função quando posso compreender melhor essas instruções como parte da função chamada. Se elas não fizerem sentido como

parte dessa função, mas devam ser chamadas com ela, basta usar *Extrair função (Extract Function)* nas instruções e na função chamada. É essencialmente o mesmo processo que descrevo a seguir, mas sem os passos para internalizar e renomear. Não é incomum fazer isso, e então, depois de refletir melhor, executar esses passos finais.

Procedimento

- Se o código repetido não estiver adjacente à chamada da função final, use *Deslocar instruções (Slide Statements)* para deixá-lo próximo.
- Se a função final só for chamada pela função original, basta cortar o código original, colá-lo na função final, testar e ignorar o restante dos passos deste procedimento.
- Se houver mais chamadas, use *Extrair função (Extract Function)* em um dos pontos de chamada para extrair tanto a chamada da função final quanto as instruções que você quer remover. Dê um nome provisório à função, mas que seja fácil de procurar.
- Converta todas as demais chamadas para que usem a nova função. Teste após cada conversão.
- Quando todas as chamadas originais estiverem usando a nova função, utilize *Internalizar função (Inline Function)* para internalizar totalmente a função original na nova função, removendo a função original.
- Use *Renomear função (Rename Function)* para modificar o nome da nova função de modo que ela tenha o mesmo nome da função original.
- Ou, se for o caso, mude para um nome melhor.

Exemplo

Começarei pelo código a seguir que gera HTML para os dados de uma foto:

```
function renderPerson(outStream, person) {
const result = [];
result.push(`<p>${person.name}</p>`);
result.push(renderPhoto(person.photo));
result.push(`<p>title: ${person.photo.title}</p>`);
result.push(emitPhotoData(person.photo));
return result.join("\n");
}
```

```
function photoDiv(p) {
  return [
    "<div>",
    `<p>title: ${p.title}</p>`,
    emitPhotoData(p),
    "</div>",
  ].join("\n");
}

function emitPhotoData(aPhoto) {
  const result = [];
  result.push(`<p>location: ${aPhoto.location}</p>`);
  result.push(`<p>date: ${aPhoto.date.toDateString()}</p>`);
  return result.join("\n");
}
```

Esse código mostra duas chamadas a emitPhotoData, cada uma precedida por uma linha de código que é semanticamente equivalente. Gostaria de remover essa duplicação movendo a exibição do título (title) para emitPhotoData. Se eu tivesse somente um ponto de chamada, bastaria cortar e colar o código, mas, quanto mais chamadas eu tiver, mais inclinado estarei a usar um procedimento mais seguro.

Começo usando *Extrair função (Extract Function)* em uma das chamadas. Extraio as instruções que quero mover para emitPhotoData, junto com a chamada ao próprio emitPhotoData.

```
function photoDiv(p) {
  return [
    "<div>",
    zznew(p),
    "</div>",
  ].join("\n");
}
function zznew(p) {
  return [
    `<p>title: ${p.title}</p>`,
    emitPhotoData(p),
  ].join("\n");
}
```

Posso agora observar as outras chamadas de emitPhotoData e, uma a uma, substituir as chamadas e as instruções anteriores por chamadas à nova função.

```
function renderPerson(outStream, person) {
  const result = [];
```

```
  result.push(`<p>${person.name}</p>`);
  result.push(renderPhoto(person.photo));
  result.push(zznew(person.photo));
  return result.join("\n");
}
```

Agora que já cuidei de todas as chamadas, uso *Internalizar função (Inline Function)* em emitPhotoData:

```
function zznew(p) {
  return [
    `<p>title: ${p.title}</p>`,
    `<p>location: ${p.location}</p>`,
    `<p>date: ${p.date.toDateString()}</p>`,
  ].join("\n");
}
```

e finalizo com *Renomear função (Rename Function)*:

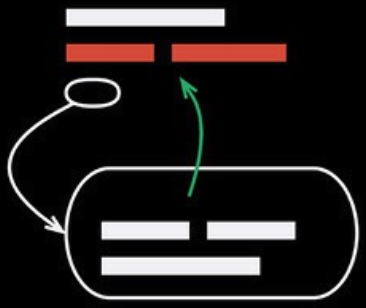
```
function renderPerson(outStream, person) {
  const result = [];
  result.push(`<p>${person.name}</p>`);
  result.push(renderPhoto(person.photo));
  result.push(emitPhotoData(person.photo));
  return result.join("\n");
}
function photoDiv(aPhoto) {
  return [
    "<div>",
    emitPhotoData(aPhoto),
    "</div>",
  ].join("\n");
}
function emitPhotoData(aPhoto) {
  return [
    `<p>title: ${aPhoto.title}</p>`,
    `<p>location: ${aPhoto.location}</p>`,
    `<p>date: ${aPhoto.date.toDateString()}</p>`,
  ].join("\n");
}
```

Também faço com que os nomes dos parâmetros estejam de acordo com minha convenção durante esse processo.

Mover instruções para os pontos de chamada

(Move Statements to Callers)

inversa de: *Mover instruções para uma função (Move Statements into Function)*



```
emitPhotoData(outStream, person.photo);

function emitPhotoData(outStream, photo) {
  outStream.write(`<p>title: ${photo.title}</p>\n`);
  outStream.write(`<p>location: ${photo.location}
</p>\n`);
}
```

```
emitPhotoData(outStream, person.photo);
outStream.write(`<p>location: ${person.photo.location}
</p>\n`);

function emitPhotoData(outStream, photo) {
  outStream.write(`<p>title: ${photo.title}</p>\n`);
}
```

Motivação

As funções são o bloco de construção básico das abstrações que implementamos como programadores. E como qualquer abstração, nem sempre definimos as fronteiras corretamente. À medida que as funcionalidades de uma base de código mudam – como ocorre na maioria dos softwares úteis –, com frequência percebemos que as fronteiras de nossas abstrações se deslocam. Para as funções, isso significa que uma unidade de comportamento que no passado era coesa e atômica se transforma em uma mistura de duas ou mais funcionalidades distintas.

Um fator que dispara esse processo se dá quando um comportamento comum usado em vários lugares deve variar em algumas de suas chamadas.

Agora temos de mover o comportamento que varia para fora da função, para os pontos de chamada. Nesse caso, uso *Deslocar instruções (Slide Statements)* para levar o comportamento que varia para o início ou para o final da função, e então utilizo *Mover instruções para os pontos de chamada*. Depois que o código variante estiver no ponto de chamada, posso alterá-lo quando necessário.

Mover instruções para os pontos de chamada funciona bem para pequenas alterações; às vezes, porém, as fronteiras entre quem chama e quem é chamado precisam ser totalmente redefinidas. Nesse caso, minha melhor opção será usar *Internalizar função (Inline Function)* e então deslocar e extrair novas funções para definir fronteiras mais apropriadas.

Procedimento

- Em circunstâncias simples, nas quais você tenha apenas uma ou duas chamadas e uma função simples sendo chamada, basta cortar a primeira linha da função chamada e colá-la (e talvez adaptá-la) nos pontos de chamada. Teste e pronto.
- Caso contrário, aplique *Extrair função (Extract Function)* em todas as instruções que você *não* queira mover; dê-lhe um nome temporário, mas fácil de procurar.

Se a função for um método sobrescrito por subclasses, faça a extração em todos os métodos de modo que o método que restar seja idêntico em todas as classes. Em seguida, remova os métodos das subclasses.

- Use *Internalizar função (Inline Function)* na função original.
- Aplique *Mudar declaração de função (Change Function Declaration)* na função extraída para renomeá-la com o nome original.

Ou dê um nome melhor para a função, se puder pensar em um.

Exemplo

Eis um caso simples: uma função com duas chamadas.

```
function renderPerson(outStream, person) {
  outStream.write(`<p>${person.name}</p>\n`);
  renderPhoto(outStream, person.photo);
  emitPhotoData(outStream, person.photo);
}
```

```
function listRecentPhotos(outStream, photos) {
  photos
    .filter(p => p.date > recentDateCutoff())
    .forEach(p => {
      outStream.write("<div>\n");
      emitPhotoData(outStream, p);
      outStream.write("</div>\n");
    });
}

function emitPhotoData(outStream, photo) {
  outStream.write(`<p>title: ${photo.title}</p>\n`);
  outStream.write(`<p>date: ${photo.date.toDateString()}</p>\n`);
  outStream.write(`<p>location: ${photo.location}</p>\n`);
}
```

Tenho de modificar o software de modo que listRecentPhotos renderize a informação de local de forma diferente, enquanto renderPerson permanece igual. Para facilitar essa mudança, usarei *Mover instruções para os pontos de chamada* na linha final.

Em geral, quando deparo com algo tão simples assim, corto a última linha de renderPerson e a colo depois das duas chamadas. No entanto, como estou explicando o que deve ser feito em casos mais complicados, executarei o procedimento mais sofisticado, porém mais seguro.

Meu primeiro passo é usar *Extrair função (Extract Function)* no código que permanecerá em emitPhotoData.

```
function renderPerson(outStream, person) {
  outStream.write(`<p>${person.name}</p>\n`);
  renderPhoto(outStream, person.photo);
  emitPhotoData(outStream, person.photo);
}

function listRecentPhotos(outStream, photos) {
  photos
    .filter(p => p.date > recentDateCutoff())
    .forEach(p => {
      outStream.write("<div>\n");
      emitPhotoData(outStream, p);
      outStream.write("</div>\n");
    });
}

function emitPhotoData(outStream, photo) {
```

```
  zztmp(outStream, photo);
  outStream.write(`<p>location: ${photo.location}</p>\n`);
}

function zztmp(outStream, photo) {
  outStream.write(`<p>title: ${photo.title}</p>\n`);
  outStream.write(`<p>date: ${photo.date.toDateString()}</p>\n`);
}
```

Em geral, o nome da função extraída será apenas temporário, portanto não me preocupo em pensar em algo significativo. No entanto, é conveniente usar algo fácil de procurar. Posso testar a essa altura, a fim de garantir que o código funcione na fronteira da chamada da função.

Agora uso *Internalizar função (Inline Function)* em uma chamada de cada vez. Começo com renderPerson.

```
function renderPerson(outStream, person) {
  outStream.write(`<p>${person.name}</p>\n`);
  renderPhoto(outStream, person.photo);
  zztmp(outStream, person.photo);
  outStream.write(`<p>location: ${person.photo.location}</p>\n`);
}
function listRecentPhotos(outStream, photos) {
  photos
    .filter(p => p.date > recentDateCutoff())
    .forEach(p => {
      outStream.write("<div>\n");
      emitPhotoData(outStream, p);
      outStream.write("</div>\n");
    });
}
function emitPhotoData(outStream, photo) {
  zztmp(outStream, photo);
  outStream.write(`<p>location: ${photo.location}</p>\n`);
}

function zztmp(outStream, photo) {
  outStream.write(`<p>title: ${photo.title}</p>\n`);
  outStream.write(`<p>date: ${photo.date.toDateString()}</p>\n`);
}
```

Testo novamente para garantir que essa chamada esteja funcionando adequadamente e então passo para a próxima.

```
function renderPerson(outStream, person) {
  outStream.write(`<p>${person.name}</p>\n`);
```

```
  renderPhoto(outStream, person.photo);
  zztmp(outStream, person.photo);
  outStream.write(`<p>location: ${person.photo.location}</p>\n`);
}
function listRecentPhotos(outStream, photos) {
  photos
    .filter(p => p.date > recentDateCutoff())
    .forEach(p => {
      outStream.write("<div>\n");
      zztmp(outStream, p);
      outStream.write(`<p>location: ${p.location}</p>\n`);
      outStream.write("</div>\n");
    });
}
function emitPhotoData(outStream, photo) {
  zztmp(outStream, photo);
  outStream.write(`<p>location: ${photo.location}</p>\n`);
}

function zztmp(outStream, photo) {
  outStream.write(`<p>title: ${photo.title}</p>\n`);
  outStream.write(`<p>date: ${photo.date.toDateString()}</p>\n`);
}
```

Em seguida, posso apagar a função externa, completando *Internalizar função (Inline Function)*.

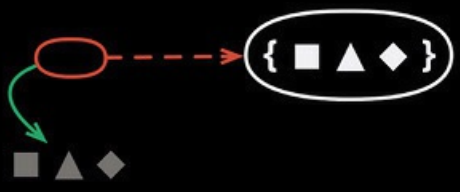
```
function renderPerson(outStream, person) {
  outStream.write(`<p>${person.name}</p>\n`);
  renderPhoto(outStream, person.photo);
  zztmp(outStream, person.photo);
  outStream.write(`<p>location: ${person.photo.location}</p>\n`);
}

function listRecentPhotos(outStream, photos) {
  photos
    .filter(p => p.date > recentDateCutoff())
    .forEach(p => {
      outStream.write("<div>\n");
      zztmp(outStream, p);
      outStream.write(`<p>location: ${p.location}</p>\n`);
      outStream.write("</div>\n");
    });
}

function emitPhotoData(outStream, photo) {
```

```
  zztmp(outStream, photo);
  outStream.write(`<p>location: ${photo.location}</p>\n`);
}

function zztmp(outStream, photo) {
  outStream.write(`<p>title: ${photo.title}</p>\n`);
  outStream.write(`<p>date: ${photo.date.toDateString()}</p>\n`);
}
```

Então renomeio zztmp de volta para o nome original.

```
function renderPerson(outStream, person) {
  outStream.write(`<p>${person.name}</p>\n`);
  renderPhoto(outStream, person.photo);
  emitPhotoData(outStream, person.photo);
  outStream.write(`<p>location: ${person.photo.location}</p>\n`);
}

function listRecentPhotos(outStream, photos) {
  photos
    .filter(p => p.date > recentDateCutoff())
    .forEach(p => {
      outStream.write("<div>\n");
      emitPhotoData(outStream, p);
      outStream.write(`<p>location: ${p.location}</p>\n`);
      outStream.write("</div>\n");
    });
}

function emitPhotoData(outStream, photo) {
  outStream.write(`<p>title: ${photo.title}</p>\n`);
  outStream.write(`<p>date: ${photo.date.toDateString()}</p>\n`);
}
```

Substituir código internalizado por chamada de função (Replace Inline Code with Function Call)



```
let appliesToMass = false;
for(const s of states) {
```

```
  if (s === "MA") appliesToMass =
true;
}
```

```
appliesToMass =
states.includes("MA");
```

Motivação

As funções me permitem empacotar comportamentos. Isso é conveniente para entender o código – uma função nomeada é capaz de explicar o propósito do código em vez de explicar o seu funcionamento. Também é importante para remover duplicações: em vez de escrever o mesmo código duas vezes, basta chamar a função. Então, caso eu precise modificar a implementação da função, não será necessário procurar códigos semelhantes para atualizar e fazer todas as mudanças. (Talvez eu tenha de olhar para as chamadas e ver se todas elas devem usar o novo código, mas isso é menos comum e mais simples.)

Se eu vir um código internalizado que faça o mesmo que uma função existente, em geral vou querer substituir esse código por uma chamada de função. Haverá uma exceção se eu considerar que a semelhança é uma coincidência – de modo que, se eu alterar o corpo da função, não espero que o comportamento desse código internalizado mude. Uma orientação a ser seguida nesse caso é o nome da função. Um bom nome deve fazer sentido no local em que estiver o código internalizado. Se o nome não fizer sentido, talvez seja um nome ruim (caso em que uso *Renomear função (Rename Function)* para corrigi-lo) ou o propósito da função é diferente daquilo que quero nesse caso – portanto essa função não deverá ser chamada.

Acho particularmente satisfatório fazer isso com chamadas de funções de biblioteca – dessa forma, não terei nem mesmo de escrever o corpo da função.

Procedimento

- Substitua o código internalizado por uma chamada à função existente.
- Teste.

Deslocar instruções (Slide Statements)

anteriormente: *Consolidar fragmentos condicionais duplicados* (Consolidate Duplicate Conditional Fragments)

```
const pricingPlan =
retrievePricingPlan();
const order = retreiveOrder();
let charge;
const chargePerUnit =
pricingPlan.unit;
```

```
const pricingPlan =
retrievePricingPlan();
const chargePerUnit =
pricingPlan.unit;
const order = retreiveOrder();
let charge;
```

Motivação

Um código é mais fácil de entender quando elementos relacionados aparecem juntos. Se várias linhas de código acessam a mesma estrutura de dados, é melhor que elas estejam juntas, em vez de estarem entrelaçadas com um código que acesse outras estruturas de dados. No caso mais simples, uso *Deslocar instruções* para manter um código como esse junto. Um caso muito comum está na declaração e no uso de variáveis. Algumas pessoas gostam de declarar todas as variáveis no início de uma função. Eu prefiro declarar a variável imediatamente antes de usá-la pela primeira vez.

Em geral, movo o código relacionado como um passo preparatório para outra refatoração – com frequência, é um *Extrair função (Extract Function)*. Colocar um código relacionado em uma função claramente distinta é uma forma de separação melhor do que apenas mover um conjunto de linhas, mas não é possível executar *Extrair função (Extract Function)* a menos que o código esteja junto antes.

Procedimento

- Identifique a posição final para a qual o fragmento será movido. Analise as instruções entre o código original e o código final para ver se há alguma interferência no fragmento candidato. Abandone a ação se houver qualquer interferência.

Um fragmento não pode ser deslocado para um ponto anterior à declaração de um elemento que o fragmento referencie.

Um fragmento não pode ser deslocado para a frente, para além de qualquer elemento que o referencie.

Um fragmento não pode ser deslocado ultrapassando qualquer instrução que modifique um elemento que ele referencie.

Um fragmento que modifique um elemento não pode ser deslocado ultrapassando qualquer outro elemento que referencie o elemento modificado.

- Corte o fragmento do código original e cole-o na posição final.
- Teste.

Se o teste falhar, experimente separar o deslocamento em passos menores. Desloque passando por menos código ou reduza a quantidade de código do fragmento que você está movendo.

Exemplo

Ao deslocar fragmentos de código, há duas decisões envolvidas: qual é o deslocamento que quero fazer, e se ele pode ser feito. A primeira decisão é muito específica de cada contexto. No nível mais simples, gosto de declarar elementos próximos dos lugares em que os uso, portanto, com frequência, desloco uma declaração para perto de seu uso. Porém, quase sempre desloco um código porque quero fazer outra refatoração – talvez reunir uma porção de código para usar *Extrair função (Extract Function)*.

Depois que eu tiver uma noção do local para onde quero mover um código, a próxima parte é decidir se posso fazê-lo. Isso envolve observar o código que estou deslocando e o código sobre o qual estou fazendo o deslocamento: eles interferem um no outro de modo a alterar o comportamento observável do programa?

Considere o fragmento de código a seguir:

```
1 const pricingPlan = retrievePricingPlan();
2 const order = retreiveOrder();
3 const baseCharge = pricingPlan.base;
4 let charge;
5 const chargePerUnit = pricingPlan.unit;
6 const units = order.units;
7 let discount;
8 charge = baseCharge + units * chargePerUnit;
9 let discountableUnits = Math.max(units - pricingPlan.discountThreshold, 0);
10 discount = discountableUnits * pricingPlan.discountFactor;
11 if (order.isRepeat) discount += 20;
12 charge = charge - discount;
13 chargeOrder(charge);
```

As sete primeiras linhas são declarações, e é relativamente fácil movê-las. Por exemplo, posso mover todo o código que trate dos descontos para que fiquem juntos, o que envolveria mover a linha 7 (`let discount`) de modo que fique acima da linha 10 (`discount = ...`). Como uma declaração não tem efeitos colaterais e não referencia nenhuma outra variável, posso mover essa linha para a frente de forma segura, até a primeira linha que referencie o próprio discount. Esse é um movimento também comum – se eu quiser usar *Extrair função (Extract Function)* na lógica de desconto, será necessário mover a declaração para baixo antes.

Faço uma análise semelhante com qualquer código que não tenha efeitos colaterais. Desse modo, posso tomar a linha 2 (`const order = ...`) e movê-la para baixo, acima da linha 6 ('const units = ...`), sem problemas.

Nesse caso, contribui o fato de que o código sobre o qual estou fazendo o deslocamento também não apresenta efeitos colaterais. Com efeito, posso reorganizar livremente um código que não tenha efeitos colaterais do jeito que eu quiser, e esse é um dos motivos de os programadores inteligentes preferirem usar um código livre de efeitos colaterais o máximo possível.

No entanto, há um porém, nesse caso. Como sei que a linha 2 não apresenta efeitos colaterais? Para ter certeza, seria necessário olhar dentro de retrieveOrder() e garantir que não há efeitos colaterais ali (e dentro de qualquer função que ela chamar, e dentro de qualquer função que suas funções chamarem, e assim por diante). Na prática, ao trabalhar com meu próprio código, sei que, de modo geral, sigo o princípio da Separação entre

Comandos-Consultas (Command-Query Separation) [mf-cps], portanto qualquer função que devolva um valor está livre de efeitos colaterais. Contudo, só posso ter certeza disso porque conheço a base de código; se estivesse trabalhando com uma base de código desconhecida, eu deveria ser mais cauteloso. Procuro seguir o princípio da Separação entre Comandos-Consultas em meu próprio código porque é muito importante saber se um código está livre de efeitos colaterais.

Ao deslocar um código que tenha um efeito colateral, ou fazer o deslocamento passando por um código com efeitos colaterais, tenho de ser muito mais cuidadoso. O que estou procurando é uma interferência entre os dois fragmentos de código. Assim, suponha que eu queira deslocar a linha 11 (`if (order.isRepeat ...`) até o final. Não posso fazer isso por causa da linha 12, pois ela referencia a variável cujo estado estou modificando na linha 11. De modo semelhante, não posso tomar a linha 13 (`chargeOrder(charge)`) e movê-la para cima porque a linha 12 modifica um estado referenciado pela linha 13. Entretanto, posso deslocar a linha 8 (`charge = baseCharge + ...`) sobre as linhas 9 a 11 porque elas não modificam nenhum estado comum.

A regra mais simples a ser seguida é que não posso deslocar um fragmento de código sobre outro se algum dado referenciado por ambos os fragmentos for modificado por um deles. Essa, porém, não é uma regra abrangente; posso deslocar tranquilamente qualquer uma das linhas a seguir, uma sobre a outra:

```
a = a + 10;
a = a + 5;
```

Entretanto, julgar se um deslocamento é seguro significa que devo realmente entender as operações envolvidas e como elas formam composições.

Como preciso me preocupar bastante com atualizações de estado, procuro removê-las o máximo possível. Assim, nesse código, eu tentaria aplicar *Separar variável (Split Variable)* em `charge` antes que eu me dê o luxo de fazer qualquer deslocamento nesse código.

Nesse caso, a análise é relativamente simples porque estou modificando apenas variáveis locais, em sua maior parte. Com estruturas de dados mais complexas, será muito mais difícil ter certeza se há interferências. Desse modo, os testes desempenham um papel importante: desloque o fragmento, execute testes, verifique se algo falhou. Se minha cobertura de testes for boa, poderei me sentir satisfeito com a refatoração. Porém, se os testes não forem

confiáveis, terei de ser mais cauteloso – ou, mais provavelmente, terei de melhorar os testes para o código no qual estou trabalhando.

A consequência mais importante de uma falha de teste após um deslocamento é fazer deslocamentos menores: em vez de deslocar o código passando por dez linhas, faço isso somente por cinco linhas, ou faço o deslocamento até uma linha que considero perigosa. Também pode significar que não vale a pena fazer o deslocamento, e que devo trabalhar em algum outro aspecto antes.

Exemplo: deslocamento com condicionais

Também posso fazer deslocamentos com condicionais. Isso envolverá a remoção de uma lógica duplicada quando faço um deslocamento para fora de uma condicional ou a adição de uma lógica duplicada quando faço o deslocamento para dentro dela.

Eis um caso em que tenho as mesmas instruções nas duas ramificações de uma condicional:

```
let result;
if (availableResources.length === 0) {
  result = createResource();
  allocatedResources.push(result);
} else {
  result = availableResources.pop();
  allocatedResources.push(result);
}
return result;
```

Posso fazer o deslocamento para fora da condicional, caso em que terei uma única instrução fora do bloco condicional.

```
let result;
if (availableResources.length === 0) {
  result = createResource();
} else {
  result = availableResources.pop();
}
allocatedResources.push(result);
return result;
```

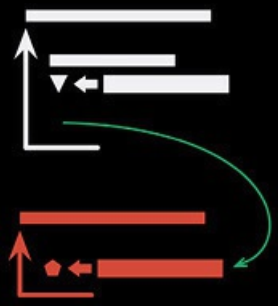
No caso inverso, deslocar um fragmento para dentro de uma condicional significa repeti-lo em cada uma das ramificações da condicional.

Leitura complementar

Já vi uma refatoração quase idêntica cujo nome é *Trocar instruções* (Swap Statement) [wake-swap]. *Trocar instruções* move fragmentos adjacentes, mas só funciona com fragmentos de uma única instrução. Você pode pensar nela como um *Deslocar instruções*, em que tanto o fragmento deslocado quanto o fragmento sobre o qual o deslocamento é feito são instruções únicas. Essa refatoração me é atraente; afinal de contas, sempre falo de executar passos pequenos – passos que podem parecer ridiculamente pequenos para aqueles a quem a refatoração é uma novidade.

Contudo, acabei escrevendo essa refatoração com fragmentos maiores porque é o que faço. Só movo uma instrução de cada vez se eu estiver tendo dificuldades com um deslocamento maior, e raramente deparo com problemas em deslocamentos maiores. Em um código mais confuso, porém, deslocamentos menores acabam sendo mais fáceis de fazer.

Dividir laço (Split Loop)



```
let averageAge = 0;
let totalSalary = 0;
for (const p of people) {
  averageAge += p.age;
  totalSalary += p.salary;
}
averageAge = averageAge /
people.length;
```

```
let totalSalary = 0;
for (const p of people) {
  totalSalary += p.salary;
}
```

```
let averageAge = 0;
for (const p of people) {
  averageAge += p.age;
}
averageAge = averageAge /
people.length;
```

Motivação

Muitas vezes, vemos laços que fazem duas tarefas distintas de uma só vez, somente porque podem fazer isso com uma única passagem por um laço. No entanto, se você estiver fazendo duas tarefas distintas no mesmo laço, sempre que for necessário modificar o laço, você terá de entender as duas tarefas. Ao separar o laço, você garante que precisará entender apenas o comportamento a ser modificado.

Dividir um laço também pode deixá-lo mais fácil de ser usado. Um laço que calcule um único valor pode simplesmente devolver esse valor. Laços que fazem muitas tarefas devem devolver estruturas ou preencher variáveis locais. Com frequência, executo uma sequência composta de *Dividir laço*, seguida de *Extrair função (Extract Function)*.

Muitos programadores se sentem desconfortáveis com essa refatoração, pois ela força você a executar o laço duas vezes. Como sempre, devo lembrar que devemos separar a refatoração da otimização (veja a seção *Refatoração e desempenho* do Capítulo 2, *página 87*). Depois que meu código estiver claro, eu o otimizo; se percorrer o laço for um gargalo, será fácil reunir os laços de novo. Porém, mesmo uma iteração por uma lista grande raramente representa um gargalo, e dividir os laços muitas vezes permite outras otimizações mais eficazes.

Procedimento

- Copie o laço.
- Identifique e elimine os efeitos colaterais duplicados.
- Teste.

Quando terminar, considere usar *Extrair função (Extract Function)* em cada laço.

Exemplo

Começarei com uma pequena porção de código que calcula o salário total (total salary) e a menor idade (youngest age).

```
  let youngest = people[0] ? people[0].age : Infinity;
  let totalSalary = 0;
  for (const p of people) {
    if (p.age < youngest) youngest = p.age;
    totalSalary += p.salary;
  }

    return `youngestAge: ${youngest}, totalSalary: ${totalSalary}`;
```

É um laço muito simples, mas ele faz dois cálculos diferentes. Para separá-los, começo copiando o laço.

```
  let youngest = people[0] ? people[0].age : Infinity;
  let totalSalary = 0;
  for (const p of people) {
    if (p.age < youngest) youngest = p.age;
    totalSalary += p.salary;
  }
  for (const p of people) {
    if (p.age < youngest) youngest = p.age;
    totalSalary += p.salary;
  }

    return `youngestAge: ${youngest}, totalSalary: ${totalSalary}`;
```

Com o laço copiado, é necessário remover a duplicação, a qual geraria resultados incorretos. Se um código no laço não tiver efeitos colaterais, eu poderia deixá-lo ali por enquanto, mas não é o caso neste exemplo.

```
  let youngest = people[0] ? people[0].age : Infinity;
  let totalSalary = 0;
  for (const p of people) {
    if (p.age < youngest) youngest = p.age;
    totalSalary += p.salary;
  }

  for (const p of people) {
    if (p.age < youngest) youngest = p.age;
    totalSalary += p.salary;
  }

  return `youngestAge: ${youngest}, totalSalary: ${totalSalary}`;
```

Oficialmente, esse é o final da refatoração *Dividir laço*. O ponto principal dessa refatoração, porém, não é o que ela faz por si só, mas a preparação que

ela permite para o próximo passo – e, em geral, estou sempre procurando extrair os laços para suas próprias funções. Usarei *Deslocar instruções (Slide Statements)* para reorganizar um pouco o código antes.

```
let totalSalary = 0;
for (const p of people) {
  totalSalary += p.salary;
}

let youngest = people[0] ? people[0].age : Infinity;
for (const p of people) {
  if (p.age < youngest) youngest = p.age;
}

return `youngestAge: ${youngest}, totalSalary: ${totalSalary}`;
```

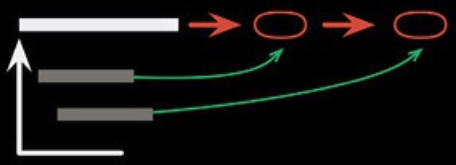
Então uso *Extrair função (Extract Function)* duas vezes.

```
return `youngestAge: ${youngestAge()}, totalSalary: ${totalSalary()}`;

function totalSalary() {
  let totalSalary = 0;
  for (const p of people) {
    totalSalary += p.salary;
  }
  return totalSalary;
}

function youngestAge() {
  let youngest = people[0] ? people[0].age : Infinity;
  for (const p of people) {
    if (p.age < youngest) youngest = p.age;
  }
  return youngest;
}
```

Quase não resisto a usar *Substituir laço por pipeline (Replace Loop with Pipeline)* para o salário total, e há um óbvio *Substituir algoritmo (Substitute Algorithm)* para o cálculo da menor idade.

```
return `youngestAge: ${youngestAge()}, totalSalary: ${totalSalary()}`;

function totalSalary() {
  return people.reduce((total,p) => total + p.salary, 0);
}
function youngestAge() {
  return Math.min(...people.map(p => p.age));
}
```

Substituir laço por pipeline (Replace Loop with Pipeline)



```
const names = [];
for (const i of input) {
  if (i.job ===
"programmer")
    names.push(i.name);
}
```

```
const names = input
  .filter(i => i.job ===
"programmer")
  .map(i => i.name)
;
```

Motivação

Assim como a maioria dos programadores, aprendi a usar laços para iterar por uma coleção de objetos. Cada vez mais, porém, os ambientes das linguagens oferecem uma construção mais apropriada: o pipeline de coleção. Os Pipelines de Coleção (Collection Pipelines) [mf-cp] permitem descrever meu processamento como uma série de operações, cada uma consumindo e gerando uma coleção. As mais comuns dessas operações são *mapear* (map), que usa uma função para transformar cada elemento da coleção de entrada, e *filtrar* (filter), que utiliza uma função para selecionar um subconjunto da coleção de entrada para passos subsequentes do pipeline. Acho que uma lógica é muito mais fácil de entender se for expressa como um pipeline – posso então ler de cima para baixo e ver como os objetos fluem pelo pipeline.

Procedimento

- Crie uma nova variável para a coleção do laço.

Pode ser uma cópia simples de uma variável existente.

- Começando de cima, para cada porção de comportamento no laço, substitua-o por uma operação do pipeline de coleção na derivação da variável de coleção do laço. Teste após cada mudança.
- Depois que todos os comportamentos forem removidos do laço, remova-o.

Se houver uma atribuição a um acumulador, atribua o resultado do pipeline ao acumulador.

Exemplo

Começarei com alguns dados: um arquivo CSV de dados sobre nossos escritórios.

```
office, country, telephone
Chicago, USA, +1 312 373 1000
Beijing, China, +86 4008 900 505
Bangalore, India, +91 80 4064 9570
Porto Alegre, Brazil, +55 51 3079 3550
Chennai, India, +91 44 660 44766

... (seguem-se mais dados)
```

A função a seguir escolhe os escritórios da Índia e devolve suas cidades e números de telefone:

```
function acquireData(input) {
  const lines = input.split("\n");
  let firstLine = true;
  const result = [];
  for (const line of lines) {
    if (firstLine) {
      firstLine = false;
      continue;
    }
    if (line.trim() === "") continue;
    const record = line.split(",");
    if (record[1].trim() === "India") {
      result.push({city: record[0].trim(), phone: record[2].trim()});
    }
  }
  return result;
}
```

Quero substituir esse laço por um pipeline de coleção.

Meu primeiro passo é criar uma variável separada para o laço trabalhar.

```
function acquireData(input) {
  const lines = input.split("\n");
  let firstLine = true;
  const result = [];
  const loopItems = lines
  for (const line of loopItems) {
    if (firstLine) {
      firstLine = false;
      continue;
    }
    if (line.trim() === "") continue;
    const record = line.split(",");
    if (record[1].trim() === "India") {
      result.push({city: record[0].trim(), phone: record[2].trim()});
    }
  }
  return result;
}
```

A primeira parte do laço está totalmente relacionada com ignorar a primeira linha do arquivo CSV. Isso pede um fatiamento, portanto removo essa primeira seção do laço e adiciono uma operação de fatiamento na definição da variável de laço.

```
function acquireData(input) {
  const lines = input.split("\n");
  let firstLine = true;
  const result = [];
  const loopItems = lines
        .slice(1);
  for (const line of loopItems) {
    if (firstLine) {
      firstLine = false;
      continue;
    }
    if (line.trim() === "") continue;
    const record = line.split(",");
    if (record[1].trim() === "India") {
      result.push({city: record[0].trim(), phone: record[2].trim()});
    }
  }
  return result;
}
```

Como bônus, isso me permite apagar `firstLine` – particularmente, gosto de

apagar variáveis de controle.

A próxima porção de comportamento remove qualquer linha em branco. Posso substituí-la por uma operação de filtrar.

```
function acquireData(input) {
  const lines = input.split("\n");
  const result = [];
  const loopItems = lines
        .slice(1)
        .filter(line => line.trim() !== "")
        ;
  for (const line of loopItems) {
    if (line.trim() === "") continue;
    const record = line.split(",");
    if (record[1].trim() === "India") {
      result.push({city: record[0].trim(), phone: record[2].trim()});
    }
  }
  return result;
}
```

Quando escrevo um pipeline, acho melhor colocar o ponto e vírgula final em sua própria linha.

Uso a operação de mapear para transformar as linhas em um array de strings – chamado confusamente de record na função original, mas é mais seguro manter o nome por enquanto e renomeá-lo depois.

```
function acquireData(input) {
  const lines = input.split("\n");
  const result = [];
  const loopItems = lines
        .slice(1)
        .filter(line => line.trim() !== "")
        .map(line => line.split(","))
        ;
  for (const line of loopItems) {
    const record = line;.split(",");
    if (record[1].trim() === "India") {
      result.push({city: record[0].trim(), phone: record[2].trim()});
    }
  }
  return result;
}
```

Filtro novamente para obter apenas os registros da Índia:

```
function acquireData(input) {
  const lines = input.split("\n");
  const result = [];
  const loopItems = lines
        .slice(1)
        .filter(line => line.trim() !== "")
        .map(line => line.split(","))
        .filter(record => record[1].trim() === "India")
        ;
  for (const line of loopItems) {
    const record = line;
    if (record[1].trim() === "India") {
      result.push({city: record[0].trim(), phone: record[2].trim()});
    }
  }
  return result;
}
```

Mapeio para o formato do registro de saída:

```
function acquireData(input) {
  const lines = input.split("\n");
  const result = [];
  const loopItems = lines
        .slice(1)
        .filter(line => line.trim() !== "")
        .map(line => line.split(","))
        .filter(record => record[1].trim() === "India")
        .map(record => ({city: record[0].trim(), phone: record[2].trim()}))
        ;
  for (const line of loopItems) {
    const record = line;
    result.push(line);
  }
  return result;
}
```

Agora, tudo o que o laço faz é atribuir valores ao acumulador. Portanto, posso removê-lo e atribuir o resultado do pipeline ao acumulador:

```
function acquireData(input) {
  const lines = input.split("\n");
  const result = lines
        .slice(1)
        .filter(line => line.trim() !== "")
```

```
        .map(line => line.split(","))
        .filter(record => record[1].trim() === "India")
        .map(record => ({city: record[0].trim(), phone: record[2].trim()}))
        ;
  for (const line of loopItems) {
    const record = line;
    result.push(line);
  }
  return result;
}
```

Essa é a parte essencial da refatoração, mas há algumas reorganizações que eu gostaria de fazer. Internalizei `result`, renomeei algumas variáveis lambda e deixei o layout mais parecido com uma tabela.

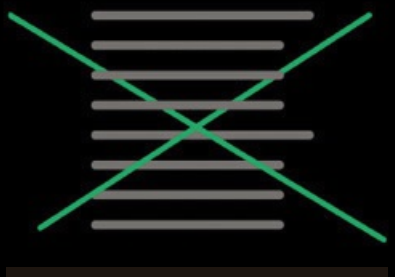
```
function acquireData(input) {
  const lines = input.split("\n");
  return lines
        .slice (1)
        .filter (line => line.trim() !== "")
        .map (line => line.split(","))
        .filter (fields => fields[1].trim() === "India")
        .map (fields => ({city: fields[0].trim(), phone: fields[2].trim()}))
        ;
}
```

Pensei em internalizar `lines` também, mas achei que sua presença explica o que está acontecendo.

Leitura complementar

Para outros exemplos de como transformar laços em pipelines, consulte meu artigo “Refactoring with Loops and Collection Pipelines” (Refatorando com laços e pipelines de coleção) [mf-ref-pipe].

Remover código morto (Remove Dead Code)



```
if(false) {
  doSomethingThatUsedToMatter();
}
```

Motivação

Quando colocamos um código no ambiente de produção, até mesmo nos dispositivos das pessoas, não somos cobrados pelo seu peso. Algumas linhas de código não usadas não deixam nossos sistemas mais lentos nem ocupam um espaço significativo de memória; na verdade, compiladores decentes as removerão instintivamente. No entanto, um código não usado continua um fardo significativo quando tentamos entender como o software funciona. Ele não porta nenhum sinal de aviso dizendo aos programadores que essa função pode ser ignorada porque não é mais chamada; assim, eles ainda terão de gastar tempo para entender o que ela faz e por que a alterar não parece alterar o resultado conforme esperado.

Quando um código deixa de ser usado, devemos apagá-lo. Não me preocupo com o fato de poder precisar dele algum dia no futuro; caso isso aconteça, tenho meu sistema de controle de versões, portanto é sempre possível recuperá-lo novamente. Se for um código que eu de fato ache que precisarei algum dia, posso colocar um comentário nele que mencione esse código perdido e em qual versão ele foi removido – honestamente, porém, sou incapaz de me lembrar de qual foi a última vez em que fiz isso, ou que tenha me arrependido por não o fazer.

Comentar um código morto já foi um hábito comum. Era conveniente na época em que os sistemas de controle de versões não eram amplamente usados ou quando eram inconvenientes. Atualmente, quando posso colocar até mesmo a menor das bases de código em um sistema de controle de versões, isso não é mais necessário.

Procedimento

- Se o código morto puder ser referenciado de fora, por exemplo, se for uma função completa, faça uma busca para verificar se há chamadas.
- Remova o código morto.

- Teste.

CAPÍTULO 9

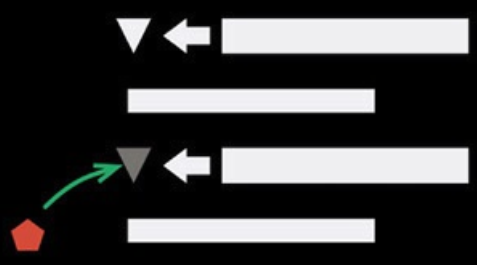
Organizando dados

As estruturas de dados desempenham um papel importante em nossos programas, portanto não é nenhuma grande surpresa que eu tenha um conjunto de refatorações com foco nelas. Um valor usado para diferentes propósitos é um terreno fértil para confusão e bugs – então, quando vejo um, utilizo *Separar variável (Split Variable)* para separar os usos. Como ocorre com qualquer elemento de programa, pensar em um nome certo para uma variável é complicado, mas importante, portanto *Renomear variável (Rename Variable)* com frequência é minha amiga. Às vezes, porém, o melhor que posso fazer com uma variável é livrar-me totalmente dela – com *Substituir variável derivada por consulta (Replace Derived Variable with Query)*.

Em geral, encontro problemas em uma base de código em virtude de uma confusão entre referências e valores, portanto uso *Mudar referência para valor (Change Reference to Value)* e *Mudar valor para referência (Change Value to Reference)* a fim de alternar entre esses estilos.

Separar variável (Split Variable)

anteriormente: *Remover atribuições a parâmetros* (Remove Assignments to Parameters)
anteriormente: *Separar variável temporária* (Split Temp)



```
let temp = 2 * (height +
width);
```

```
console.log(temp);
temp = height * width;
console.log(temp);
```

```
const perimeter = 2 * (height +
width);
console.log(perimeter);
const area = height * width;
console.log(area);
```

Motivação

As variáveis têm muitos usos. Alguns desses usos resultam naturalmente em uma variável receber valores diversas vezes. Variáveis de laço mudam a cada execução do laço (por exemplo, `i` em `for (let i=0; i<10; i++)`). Variáveis acumuladoras armazenam um valor calculado durante o método.

Muitas outras variáveis são usadas para armazenar o resultado de uma porção de código extensa a fim de facilitar futuras referências. Esses tipos de variáveis devem receber valor somente uma vez. Se receberem mais de uma vez, é sinal de que elas têm mais de uma responsabilidade no método. Qualquer variável com mais de uma responsabilidade deve ser substituída por diversas variáveis, uma para cada responsabilidade. Usar uma variável para duas tarefas distintas é muito confuso para o leitor.

Procedimento

- Altere o nome da variável em sua declaração e na primeira atribuição.

Se as atribuições subsequentes estiverem no formato `i = i + algo`, ela é uma variável acumuladora, portanto não a separe. Uma variável acumuladora com frequência é usada para calcular somas, concatenar strings, escrever em um stream ou fazer acréscimos em uma coleção.

- Se for possível, declare a nova variável como imutável.
- Mude todas as referências à variável até sua segunda atribuição.
- Teste.
- Repita em etapas, a cada etapa renomeando a variável na declaração e mudando as referências até a próxima atribuição; faça isso até alcançar a última atribuição.

Exemplo

Neste exemplo, calcularei a distância percorrida por um haggis[1]. De um ponto de partida, um haggis é submetido a uma força inicial. Após um tempo, uma força secundária é aplicada para acelerar mais o haggis. Usando as leis comuns de movimento, posso calcular a distância percorrida da seguinte maneira:

```
function distanceTravelled (scenario, time) {
  let result;
  let acc = scenario.primaryForce / scenario.mass;
  let primaryTime = Math.min(time, scenario.delay);
  result = 0.5 * acc * primaryTime * primaryTime;
  let secondaryTime = time - scenario.delay;
  if (secondaryTime > 0) {
    let primaryVelocity = acc * scenario.delay;
    acc = (scenario.primaryForce + scenario.secondaryForce) / scenario.mass;
    result += primaryVelocity * secondaryTime + 0.5 * acc * secondaryTime * secondaryTime;
  }
  return result;
}
```

Uma bela funçãozinha confusa. O aspecto interessante em nosso exemplo é o modo como a variável `acc` é definida duas vezes. Ela tem duas responsabilidades: uma, que consiste em armazenar a aceleração inicial da primeira força, e outra para armazenar a aceleração resultante das duas forças. Gostaria de separar essa variável.

Ao tentar entender como uma variável é usada, será conveniente se meu editor puder destacar todas as ocorrências de um símbolo dentro de uma função ou de um arquivo. A maioria dos editores modernos é capaz de fazer isso facilmente.

Começo no início, mudando o nome da variável e declarando o novo nome como `const`. Em seguida, modifico todas as referências à variável a partir desse ponto até a próxima atribuição. Na próxima atribuição, eu a declaro:

```
function distanceTravelled (scenario, time) {
  let result;
  const primaryAcceleration = scenario.primaryForce / scenario.mass;
  let primaryTime = Math.min(time, scenario.delay);
  result = 0.5 * primaryAcceleration * primaryTime * primaryTime;
  let secondaryTime = time - scenario.delay;
```

```
  if (secondaryTime > 0) {
    let primaryVelocity = primaryAcceleration * scenario.delay;
    let acc = (scenario.primaryForce + scenario.secondaryForce) / scenario.mass;
    result += primaryVelocity * secondaryTime + 0.5 * acc * secondaryTime * secondaryTime;
  }
  return result;
}
```

Escolho o novo nome para representar apenas o primeiro uso da variável. Deixo-a como const para garantir que ela receba valor somente uma vez. Posso então declarar a variável original em sua segunda atribuição. Agora posso compilar e testar, e tudo deverá estar funcionando.

Continuo a partir da segunda atribuição à variável. Removo totalmente o nome da variável original, substituindo-a por um novo nome de variável que esteja de acordo com o segundo uso.

```
function distanceTravelled (scenario, time) {
  let result;
  const primaryAcceleration = scenario.primaryForce / scenario.mass;
  let primaryTime = Math.min(time, scenario.delay);
  result = 0.5 * primaryAcceleration * primaryTime * primaryTime;
  let secondaryTime = time - scenario.delay;
  if (secondaryTime > 0) {
    let primaryVelocity = primaryAcceleration * scenario.delay;
    const secondaryAcceleration =
        (scenario.primaryForce + scenario.secondaryForce) / scenario.mass;
    result += primaryVelocity * secondaryTime +
      0.5 * secondaryAcceleration * secondaryTime * secondaryTime;
  }
  return result;
}
```

Tenho certeza de que você é capaz de pensar em várias outras refatorações a serem feitas nesse caso. Aproveite. (Garanto que é melhor do que comer o haggis – você sabe o que eles colocam nele?)

Exemplo: fazendo atribuições a um parâmetro de entrada

Outro caso de separação de uma variável é aquele em que a variável é declarada como um parâmetro de entrada. Considere um código como este:

```
function discount (inputValue, quantity) {
  if (inputValue > 50) inputValue = inputValue - 2;
```

```
  if (quantity > 100) inputValue = inputValue - 1;
  return inputValue;
}
```

Nesse caso, inputValue é usado tanto com o intuito de fornecer uma entrada para a função quanto a fim de armazenar o resultado para quem fez a chamada. (Como JavaScript tem parâmetros passados por valor, nenhuma modificação em inputValue será vista por quem chamou.)

Nessa situação, eu separaria essa variável.

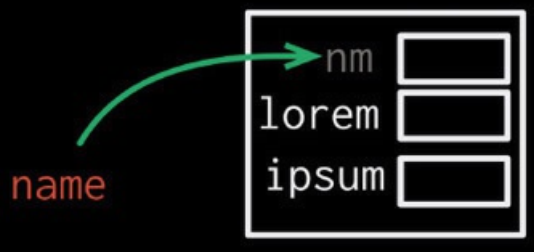
```
function discount (originalInputValue, quantity) {
  let inputValue = originalInputValue;
  if (inputValue > 50) inputValue = inputValue - 2;
  if (quantity > 100) inputValue = inputValue - 1;
  return inputValue;
}
```

Então uso *Renomear variável (Rename Variable)* duas vezes para criar nomes melhores.

```
function discount (inputValue, quantity) {
  let result = inputValue;
  if (inputValue > 50) result = result - 2;
  if (quantity > 100) result = result - 1;
  return result;
}
```

Você perceberá que mudei a segunda linha para usar inputValue como a fonte do dado. Embora ambos sejam iguais, acho que essa linha na verdade tem a ver com aplicar a modificação no valor do resultado baseado no valor de entrada original, e não no valor (que, coincidentemente, é o mesmo) do acumulador resultante.

Renomear campo (Rename Field)




```
class
Organization {
```

```
  get name() {...}
}
```

```
class
Organization {
  get title() {...}
}
```

Motivação

Nomes são importantes, e nomes de campo em estruturas de registro podem ser especialmente importantes quando essas estruturas são usadas de forma ampla em um programa. As estruturas de dados desempenham um papel particularmente importante para a compreensão. Muitos anos atrás, Fred Brooks disse: “Mostre-me seus fluxogramas e esconda suas tabelas, e eu continuarei atônito. Mostre-me suas tabelas e, em geral, não precisarei de seus fluxogramas; elas serão óbvias”. Embora eu não veja muitas pessoas desenhando fluxogramas atualmente, a citação permanece válida. As estruturas de dados são o segredo para entender o que está acontecendo.

Por serem tão importantes, é essencial manter essas estruturas de dados organizadas. Como tudo, minha compreensão dos dados melhora quanto mais eu trabalhar com o software, portanto é crucial que essa compreensão melhorada seja embutida no programa.

Você talvez queira renomear um campo em uma estrutura de registro, mas a ideia também se aplica a classes. Métodos getter e setter formam um campo eficaz para os usuários da classe. Renomeá-los é tão importante quanto renomear campos em estruturas de registro simples.

Procedimento

- Se o registro tiver um escopo limitado, renomeie todos os acessos ao campo e teste; não há necessidade de executar o restante do procedimento.
- Se o registro ainda não estiver encapsulado, aplique *Encapsular registro (Encapsulate Record)*.
- Renomeie o campo privado no objeto, faça adaptações nos métodos internos para que se tornem apropriados.

- Teste.
- Se o construtor usar o nome, aplique *Mudar declaração de função (Change Function Declaration)* para renomeá-lo.
- Aplique *Renomear função (Rename Function)* nos métodos de acesso.

Exemplo: renomear um campo

Começarei com uma constante.

```
const organization = {name: "Acme Gooseberries", country: "GB"};
```

Quero mudar “name” (nome) para “title” (título). O objeto é amplamente usado na base de código, e há atualizações do título no código. Desse modo, meu primeiro passo é aplicar *Encapsular registro (Encapsulate Record)*.

```
class Organization {
  constructor(data) {
    this._name = data.name;
    this._country = data.country;
  }
  get name() {return this._name;}
  set name(aString) {this._name = aString;}
  get country() {return this._country;}
  set country(aCountryCode) {this._country = aCountryCode;}
}

const organization = new Organization({name: "Acme Gooseberries", country: "GB"});
```

Agora que encapsulei a estrutura de registro na classe, há quatro lugares que preciso considerar para renomear: a função de leitura, a função de escrita, o construtor e a estrutura de dados interna. Embora pareça que eu aumentei minha carga de trabalho, na verdade, isso simplifica minha tarefa, pois posso agora modificar esses pontos de modo independente, em vez de fazê-lo de uma só vez, executando passos menores. Passos menores significam menos erros a cada passo – portanto, menos trabalho. Não seria menos trabalho se eu jamais cometesse erros – mas não cometer erros é uma fantasia da qual eu já abri mão há muito tempo.

Como copiei a estrutura de dados de entrada para a estrutura de dados interna, preciso separá-las para que eu trabalhe com elas de modo independente. Posso fazer isso definindo um campo separado e adaptando o construtor e os métodos de acesso para que o utilizem.

class Organization…

```
class Organization {
  constructor(data) {
    this._title = data.name;
    this._country = data.country;
  }
  get name() {return this._title;}
  set name(aString) {this._title = aString;}
  get country() {return this._country;}
  set country(aCountryCode) {this._country = aCountryCode;}
}
```

Em seguida, acrescento o suporte para usar “title” no construtor.

class Organization…

```
class Organization {
  constructor(data) {
    this._title = (data.title !== undefined) ? data.title : data.name;
    this._country = data.country;
  }
  get name() {return this._title;}
  set name(aString) {this._title = aString;}
  get country() {return this._country;}
  set country(aCountryCode) {this._country = aCountryCode;}
}
```

Agora quem chama meu construtor pode usar o nome ou o título (com o título tendo precedência). Posso agora verificar todas as chamadas ao construtor e alterá-las uma a uma para que usem o novo nome.

```
const organization = new Organization({title: "Acme Gooseberries", country: "GB"});
```

Depois que tudo isso for feito, removo o suporte para o nome.

class Organization…

```
class Organization {
  constructor(data) {
    this._title = data.title;
    this._country = data.country;
  }
  get name() {return this._title;}
  set name(aString) {this._title = aString;}
  get country() {return this._country;}
  set country(aCountryCode) {this._country = aCountryCode;}
}
```

Agora que o construtor e o dado usam o novo nome, posso modificar os métodos de acesso; é simples, e basta aplicar *Renomear função (Rename Function)* em cada um.

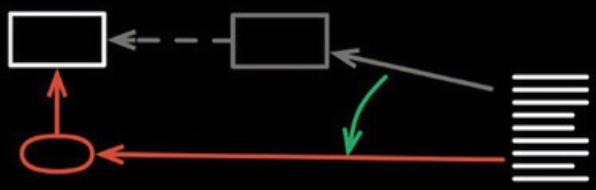
class Organization…

```
class Organization {
  constructor(data) {
    this._title = data.title;
    this._country = data.country;
  }
  get title() {return this._title;}
  set title(aString) {this._title = aString;}
  get country() {return this._country;}
  set country(aCountryCode) {this._country = aCountryCode;}
}
```

Mostrei esse processo em sua forma mais pesada, necessária no caso de uma estrutura de dados amplamente usada. Se ela for usada apenas localmente, como em uma única função, é provável que poderei apenas renomear as diversas propriedades de uma só vez, sem fazer o encapsulamento. É uma questão de avaliar quando aplicar o procedimento completo – no entanto, como sempre na refatoração, se meus testes falharem, é sinal de que devo usar o procedimento gradual.

Algumas linguagens permitem deixar uma estrutura de dados imutável. Nesse caso, em vez de encapsulá-la, posso copiar o valor para o novo nome, alterando gradualmente os usuários e, então, renomeando o nome antigo. Duplicar dados é uma receita para desastre com estruturas de dados mutáveis; impedir esses desastres é o motivo pelo qual os dados imutáveis são tão populares.

Substituir variável derivada por consulta (Replace Derived Variable with Query)



```
get discountedTotal() {return
```

```
  this._discountedTotal;}
  set discount(aNumber) {
    const old = this._discount;
    this._discount = aNumber;
    this._discountedTotal += old - aNumber;
  }
```

```
get discountedTotal() {return this._baseTotal -
this._discount;}
set discount(aNumber) {this._discount = aNumber;}
```

Motivação

Uma das maiores fontes de problemas em software são os dados mutáveis. Mudanças em dados muitas vezes podem acoplar porções de código de modo inconveniente, com mudanças em uma parte resultando em efeitos em cascata difíceis de identificar. Em muitas situações, remover totalmente os dados mutáveis não é realista – mas defendo minimizar o máximo possível o escopo desses dados.

Uma maneira de causar um grande impacto é remover qualquer variável que seja possível calcular facilmente. Um cálculo muitas vezes deixa mais claro o significado dos dados, e estará protegido de ser corrompido se você falhar em atualizar a variável conforme os dados originais mudarem.

Uma exceção razoável é quando o dado original para o cálculo é imutável e podemos forçar para que o resultado seja imutável também. Operações de transformação que criam novas estruturas de dados são, desse modo, razoáveis de serem mantidas como estão, mesmo se puderem ser substituídas por cálculos. Com efeito, há uma dualidade, nesse caso, entre objetos que encapsulam uma estrutura de dados e uma série de propriedades calculadas e funções que transformam uma estrutura de dados em outra. A alternativa com um objeto será claramente melhor se os dados originais mudarem e você tiver de administrar o tempo de vida das estruturas de dados derivadas. Todavia, se os dados originais forem imutáveis, ou os dados derivados forem muito transientes, as duas abordagens serão eficazes.

Procedimento

- Identifique todos os pontos de atualização da variável. Se necessário, use

Separar variável (Split Variable) para separar cada ponto de atualização.

- Crie uma função que calcule o valor da variável.
- Use *Introduzir asserção (Introduce Assertion)* para garantir que a variável e o cálculo tenham o mesmo resultado sempre que a variável for usada.

Se necessário, use *Encapsular variável (Encapsulate Variable)* para fornecer um lar para a asserção.

- Teste.
- Substitua qualquer leitura da variável por uma chamada à nova função.
- Teste.
- Aplique *Remover código morto (Remove Dead Code)* na declaração e nas atualizações da variável.

Exemplo

Eis um exemplo pequeno, porém perfeitamente concebido, de um código feio:

class ProductionPlan…

```
get production() {return this._production;}
applyAdjustment(anAdjustment) {
  this._adjustments.push(anAdjustment);
  this._production += anAdjustment.amount;
}
```

A feiura está nos olhos de quem vê; nesse caso, vejo feiura na duplicação – não é uma duplicação comum de código, mas uma duplicação de dados. Quando aplico uma correção (adjustment), não estou apenas armazenando essa correção, mas uso-a também para modificar um acumulador. Posso simplesmente calcular esse valor, sem ter de atualizá-lo.

No entanto, sou um sujeito cauteloso. Parto da hipótese de que posso apenas o calcular – é possível testar essa hipótese usando *Introduzir asserção (Introduce Assertion)*.

class ProductionPlan…

```
get production() {
  assert(this._production === this.calculatedProduction);
  return this._production;
}
```

```
get calculatedProduction() {
  return this._adjustments
    .reduce((sum, a) => sum + a.amount, 0);
}
```

Com a asserção definida, executo meus testes. Se a asserção não falhar, posso substituir a devolução do campo pela devolução do cálculo:

class ProductionPlan…

```
get production() {
  assert(this._production === this.calculatedProduction);
  return this.calculatedProduction;
}
```

Em seguida, uso *Internalizar função (Inline Function)*.

class ProductionPlan…

```
get production() {
  return this._adjustments
    .reduce((sum, a) => sum + a.amount, 0);
}
```

Limpo qualquer referência à antiga variável usando *Remover código morto (Remove Dead Code)*:

class ProductionPlan…

```
applyAdjustment(anAdjustment) {
  this._adjustments.push(anAdjustment);
  this._production += anAdjustment.amount;
}
```

Exemplo: mais de uma origem

O exemplo anterior é bom e simples porque há claramente uma única origem para o valor de `production`. Às vezes, porém, mais de um elemento pode ser combinado no acumulador.

class ProductionPlan…

```
constructor (production) {
  this._production = production;
  this._adjustments = [];
}
get production() {return this._production;}
applyAdjustment(anAdjustment) {
  this._adjustments.push(anAdjustment);
```

```
    this._production += anAdjustment.amount;
  }
```

Se eu usar o mesmo *Introduzir asserção (Introduce Assertion)* anterior, teria agora uma falha para qualquer caso em que o valor inicial da produção não seja zero.

Contudo, ainda posso substituir os dados derivados. A única diferença é que devo aplicar *Separar variável (Split Variable)* antes.

```
  constructor (production) {
    this._initialProduction = production;
    this._productionAccumulator = 0;
    this._adjustments = [];
  }
  get production() {
    return this._initialProduction + this._productionAccumulator;
  }
```

Agora posso usar *Introduzir asserção (Introduce Assertion)*:

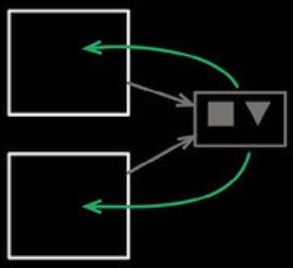
class ProductionPlan…

```
  get production() {
    assert(this._productionAccumulator === this.calculatedProductionAccumulator);
    return this._initialProduction + this._productionAccumulator;
  }
  get calculatedProductionAccumulator() {
    return this._adjustments
      .reduce((sum, a) => sum + a.amount, 0);
  }
```

e prosseguir basicamente do mesmo modo que antes. Entretanto, eu estaria inclinado a deixar totalProductionAjustments como uma propriedade, sem internalizá-la.

Mudar referência para valor (Change Reference to Value)

inversa de: *Mudar valor para referência (Change Value to Reference)*



```
class Product {
  applyDiscount(arg) {this._price.amount -=
arg;}
```

```
class Product {
  applyDiscount(arg) {
    this._price = new Money(this._price.amount - arg,
this._price.currency);
  }
```

Motivação

Quando deixo um objeto, ou uma estrutura de dados, aninhado em outro, posso tratar o objeto interno como uma referência ou como um valor. A diferença é mais evidentemente visível no modo como trato as atualizações nas propriedades do objeto interno. Se eu tratar esse objeto interno como uma referência, atualizarei a sua propriedade, mantendo o mesmo objeto interno. Se tratá-lo como um valor, substituirei o objeto interno completo por um novo objeto com a propriedade desejada.

Se eu tratar um campo como um valor, posso mudar a classe do objeto interno e transformá-lo em um Objeto de Valor (Value Object) [mf-vo]. De modo geral, é mais fácil raciocinar com objetos de valor, particularmente porque eles são imutáveis. Em geral, é mais fácil lidar com estruturas de dados imutáveis. Posso passar um dado imutável para outras partes do programa e não me preocupar com o fato de ele poder mudar sem que o objeto que o inclua tenha ciência da mudança. Posso replicar valores em meu programa sem me preocupar em manter ligações de memória. Objetos de valor são particularmente convenientes em sistemas distribuídos e concorrentes.

Isso também sugere quando não devo fazer essa refatoração. Se eu quiser compartilhar um objeto entre vários objetos de modo que qualquer alteração

no objeto compartilhado seja visível a todos os colaboradores, será necessário que o objeto compartilhado seja uma referência.

Procedimento

- Verifique se a classe candidata é imutável ou pode se tornar imutável.
- Para cada setter, aplique *Remover método de escrita (Remove Setting Method)*.
- Forneça um método de igualdade baseado em valor que utilize os campos do objeto de valor.

A maioria dos ambientes de linguagens oferece uma função de igualdade que pode ser sobrescrita para essa finalidade. Em geral, você deve sobrescrever um método gerador de código hash também.

Exemplo

Suponha que temos um objeto pessoa que armazene um número de telefone bruto.

class Person…

```
constructor() {
  this._telephoneNumber = new TelephoneNumber();
}
get officeAreaCode() {return this._telephoneNumber.areaCode;}
set officeAreaCode(arg) {this._telephoneNumber.areaCode = arg;}
get officeNumber() {return this._telephoneNumber.number;}
set officeNumber(arg) {this._telephoneNumber.number = arg;}
```

class TelephoneNumber…

```
get areaCode() {return this._areaCode;}
set areaCode(arg) {this._areaCode = arg;}
get number() {return this._number;}
set number(arg) {this._number = arg;}
```

Essa situação é o resultado de um *Extrair classe (Extract Class)*, em que o pai antigo continua armazenando métodos de atualização para o novo objeto. É uma boa hora para aplicar *Mudar referência para valor*, pois há apenas uma referência à nova classe.

A primeira tarefa a ser feita é deixar o número de telefone imutável. Faço isso aplicando *Remover método de escrita (Remove Setting Method)* nos

campos. O primeiro passo dessa refatoração é usar *Mudar declaração de função (Change Function Declaration)* para adicionar dois campos no construtor e melhorá-lo de modo que ele chame os setters.

class TelephoneNumber…

```
constructor(areaCode, number) {
  this._areaCode = areaCode;
  this._number = number;
}
```

Agora, observo quem chama os setters. Para cada chamada, tenho de modificá-la para uma nova atribuição. Começo pelo código de área.

class Person…

```
get officeAreaCode() {return this._telephoneNumber.areaCode;}
set officeAreaCode(arg) {
  this._telephoneNumber = new TelephoneNumber(arg, this.officeNumber);
}
get officeNumber() {return this._telephoneNumber.number;}
set officeNumber(arg) {this._telephoneNumber.number = arg;}
```

Repito então esse passo com o campo restante.

class Person…

```
get officeAreaCode() {return this._telephoneNumber.areaCode;}
set officeAreaCode(arg) {
  this._telephoneNumber = new TelephoneNumber(arg, this.officeNumber);
}
get officeNumber() {return this._telephoneNumber.number;}
set officeNumber(arg) {
  this._telephoneNumber = new TelephoneNumber(this.officeAreaCode, arg);
}
```

Agora que é imutável, o número do telefone está pronto para se tornar um valor de verdade. O teste de cidadania para um objeto de valor é ver se ele usa uma igualdade baseada em valor. Essa é uma área em que JavaScript deixa a desejar, pois não há nada na linguagem nem nas bibliotecas principais que entenda substituir uma igualdade baseada em referência por uma baseada em valor. O melhor que posso fazer é criar meu próprio método equals.

class TelephoneNumber…

```
equals(other) {
  if (!(other instanceof TelephoneNumber)) return false;
  return this.areaCode === other.areaCode &&
```

```
    this.number === other.number;
}
```

Também é importante testar o código com algo como:

```
it('telephone equals', function() {
  assert( new TelephoneNumber("312", "555-0142")
        .equals(new TelephoneNumber("312", "555-0142")));
});
```

A formatação incomum que uso nesse caso deve deixar claro que são a mesma chamada de construtor.

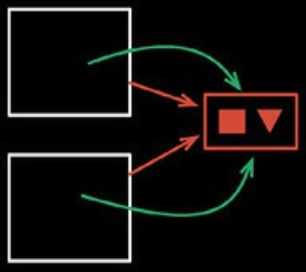
A tarefa essencial no teste é criar dois objetos independentes e testar se são iguais.

Na maioria das linguagens orientadas a objetos, há um teste de igualdade embutido que se espera que seja sobrescrito para uma igualdade baseada em valor. Em Ruby, posso sobrescrever o operador `==`; em Java, sobrescrevo o método `Object.equals()`. Sempre que sobrescrevo um método de igualdade, em geral preciso sobrescrever também um método de geração de código hash (por exemplo, `Object.hashCode()` em Java) a fim de garantir que as coleções que usam hashing funcionem de modo apropriado com meu novo valor.

Se o número de telefone for usado por mais de um cliente, o procedimento continuará o mesmo. Ao aplicar *Remover método de escrita (Remove Setting Method)*, estarei modificando vários clientes em vez de um só. Também vale a pena ter testes para números de telefone diferentes, assim como comparações com números que não são de telefone e valores nulos.

Mudar valor para referência (Change Value to Reference)

inversa de: *Mudar referência para valor (Change Reference to Value)*



let customer = new

```
Customer(customerData);
```

```
let customer =
customerRepository.get(customerData.id);
```

Motivação

Uma estrutura de dados pode ter vários registros associados à mesma estrutura de dados lógica. Posso ler uma lista de pedidos, alguns dos quais são para o mesmo cliente. Quando tenho um compartilhamento como esse, posso representá-lo tratando o cliente como um valor ou como uma referência. Com um valor, os dados do cliente serão copiados em cada pedido; com uma referência, há apenas uma estrutura de dados à qual vários pedidos estarão associados.

Se o cliente jamais tiver de ser atualizado, as duas abordagens serão razoáveis. Talvez seja um pouco confuso ter várias cópias dos mesmos dados, mas é suficientemente comum a ponto de não ser um problema. Em alguns casos, pode haver problemas com memória em consequência das várias cópias – mas, como qualquer problema de desempenho, isso é relativamente raro.

A maior dificuldade em ter cópias físicas dos mesmos dados lógicos ocorre quando é necessário atualizar os dados compartilhados. Tenho então de encontrar todas as cópias e atualizá-las. Se eu me esquecer de uma, terei uma inconsistência problemática em meus dados. Nesse caso, muitas vezes vale a pena mudar os dados copiados para uma única referência. Desse modo, qualquer alteração será visível a todos os pedidos do cliente.

Mudar um valor para uma referência resulta em apenas um objeto presente para uma entidade, e em geral significa que preciso de algum tipo de repositório no qual eu acesse esses objetos. Então crio o objeto para uma entidade somente uma vez, e, em todos os demais lugares, eu o obtenho a partir do repositório.

Procedimento

- Crie um repositório para instâncias do objeto relacionado (caso ainda não haja um presente).

- Certifique-se de que o construtor tenha uma forma de consultar a instância correta do objeto relacionado.
- Modifique os construtores para que o objeto que armazena o dado utilize o repositório a fim de obter o objeto relacionado. Teste após cada mudança.

Exemplo

Começarei com uma classe que representa pedidos (orders); estes podem ser criados a partir de um documento JSON de entrada. Parte dos dados do pedido consiste de um ID do cliente a partir do qual crio um objeto cliente (customer).

class Order…

```
constructor(data) {
  this._number = data.number;
  this._customer = new Customer(data.customer);
  // carrega outros dados
}
get customer() {return this._customer;}
```

class Customer…

```
constructor(id) {
  this._id = id;
}
get id() {return this._id;}
```

O objeto cliente que crio dessa forma é um valor. Se eu tiver cinco pedidos que referenciem o ID 123 do cliente, terei cinco objetos cliente separados. Qualquer modificação que eu fizer em um deles não se refletirá nos demais. Caso eu queira enriquecer os objetos cliente, talvez coletando dados de um serviço de atendimento ao cliente, será necessário atualizar todos os cinco clientes com os mesmos dados. Ter objetos duplicados desse modo sempre me deixa nervoso – é confuso ter vários objetos que representam a mesma entidade, por exemplo, um cliente. Esse problema será particularmente inconveniente quando o objeto cliente for mutável, o que pode resultar em inconsistências entre os objetos cliente.

Se eu quiser usar sempre o mesmo objeto cliente, precisarei de um lugar para armazená-lo. Exatamente onde armazenar entidades como essas variará de aplicação para aplicação, mas, em um caso simples, gosto de usar um objeto de repositório (repository object) [mf-repos].

```
let _repositoryData;

export function initialize() {
  _repositoryData = {};
  _repositoryData.customers = new Map();
}

export function registerCustomer(id) {
  if (! _repositoryData.customers.has(id))
    _repositoryData.customers.set(id, new Customer(id));
  return findCustomer(id);
}

export function findCustomer(id) {
  return _repositoryData.customers.get(id);
}
```

O repositório permite que eu registre objetos cliente com um ID e garante que criarei somente um objeto cliente com o mesmo ID. Com isso definido, posso modificar o construtor do pedido de modo a usar o repositório.

Muitas vezes, ao fazer essa refatoração, o repositório já existirá, portanto posso apenas usá-lo.

O próximo passo é descobrir como o construtor do pedido pode obter o objeto cliente correto. Nesse caso é fácil, pois o ID do cliente está presente no stream de dados de entrada.

class Order…

```
constructor(data) {
  this._number = data.number;
  this._customer = registerCustomer(data.customer);
  // carrega outros dados
}
get customer() {return this._customer;}
```

Agora, qualquer alteração que eu fizer no cliente de um pedido será sincronizada entre todos os pedidos que compartilhem o mesmo cliente.

Neste exemplo, criei um novo objeto cliente com o primeiro pedido que o referencia. Outra abordagem comum é obter uma lista de clientes, preencher o repositório com eles e, então, fazer uma associação à medida que eu ler os pedidos. Nesse caso, um pedido que contenha um ID de cliente que não esteja no repositório sinalizaria um erro.

Um problema com esse código é que o corpo do construtor está acoplado ao repositório global. Os globais devem ser tratados com cuidado – assim como

um medicamento poderoso, eles podem ser benéficos em pequenas doses, mas serão um veneno caso usados em excesso. Se eu estiver preocupado com isso, posso passar o repositório como um parâmetro para o construtor.

---

1 N.T.: *Haggis* é o nome que se dá a um prato típico escocês feito com estômago de carneiro, recheado com miúdos, aveia e especiarias.

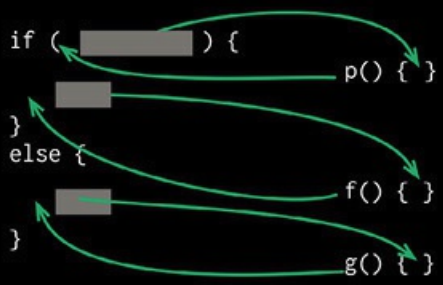
CAPÍTULO 10

Simplificando lógicas condicionais

Boa parte da eficácia dos programas provém de sua capacidade de implementar lógicas condicionais – infelizmente, porém, muito da complexidade dos programas se encontra nessas condicionais. Com frequência, uso a refatoração para deixar as seções com condicionais mais fáceis de entender. Aplico regularmente *Decompor condicional (Decompose Conditional)* para condicionais complicadas, e uso *Consolidar expressão condicional (Consolidate Conditional Expression)* para deixar as combinações lógicas mais claras. Uso *Substituir condicional aninhada por cláusulas de guarda (Replace Nested Conditional with Guard Claus* para deixar mais claros os casos em que gostaria de executar algumas verificações prévias ao meu processamento principal. Se vejo muitas condições usando a mesma lógica de alternância, é uma boa hora para lançar mão de *Substituir condicional por polimorfismo (Replace Conditional with Polymorphism)*.

Muitas condicionais são usadas para tratar casos especiais, por exemplo, os nulos; se boa parte dessa lógica for a mesma, então *Introduzir caso especial (Introduce Special Case)* – muitas vezes chamada de *Introduzir objeto nulo (Introduce Null Object)* – é capaz de remover muito código duplicado. Embora eu goste muito de remover condições, se eu quiser informar (e conferir) o estado de um programa, acho que *Introduzir asserção (Introduce Assertion)* é um acréscimo que vale a pena fazer.

Decompor condicional (Decompose Conditional)




```
if (!aDate.isBefore(plan.summerStart) &&
!aDate.isAfter(plan.summerEnd))
  charge = quantity * plan.summerRate;
else
  charge = quantity * plan.regularRate + plan.regularServiceCharge;
```

```
if (summer())
  charge =
summerCharge();
else
  charge =
regularCharge();
```

Motivação

Uma das fontes mais comuns de complexidade em um programa é uma lógica condicional complexa. À medida que escrevo código para várias situações que dependam de diversas condições, posso acabar rapidamente com uma função bem longa. O tamanho de uma função, por si só, é um fator que dificulta a leitura, mas as condições aumentam a dificuldade. O problema em geral está no fato de o código, tanto nas verificações das condições quanto nas ações, me dizer o que acontece, mas facilmente encobrir *por que* acontece.

Assim como em qualquer bloco de código grande, posso deixar minha intenção mais clara decompondo-o e substituindo cada porção de código por uma chamada de função cujo nome seja atribuído com base na intenção da respectiva porção. Com as condições, gosto particularmente de fazer isso para a parte condicional e para cada uma das alternativas. Dessa forma, enfatizo a condição e deixo claro o código com base no qual se dá a ramificação. Também enfatizo o motivo para a ramificação.

Na verdade, esse é um caso particular da aplicação de *Extrair função (Extract Function)* em meu código, mas gosto de destacá-lo, pois, com frequência, vejo aí um motivo muito relevante para usar essa refatoração.

Procedimento

- Aplique *Extrair função (Extract Function) na condição e em cada ramo da condicional.*

Exemplo

Suponha que eu esteja calculando o valor cobrado por algo que tenha preços distintos no inverno e no verão (summer):

```
if (!aDate.isBefore(plan.summerStart) && !aDate.isAfter(plan.summerEnd))
  charge = quantity * plan.summerRate;
else
  charge = quantity * plan.regularRate + plan.regularServiceCharge;
```

Extraio a condição, colocando-a em sua própria função.

```
if (summer())
  charge = quantity * plan.summerRate;
else
  charge = quantity * plan.regularRate + plan.regularServiceCharge;

function summer() {
  return !aDate.isBefore(plan.summerStart) && !aDate.isAfter(plan.summerEnd);
}
```

Então cuido do ramo then:

```
if (summer())
  charge = summerCharge();
else
  charge = quantity * plan.regularRate + plan.regularServiceCharge;

function summer() {
  return !aDate.isBefore(plan.summerStart) && !aDate.isAfter(plan.summerEnd);
}
function summerCharge() {
  return quantity * plan.summerRate;
}
```

Por fim, trato o ramo else:

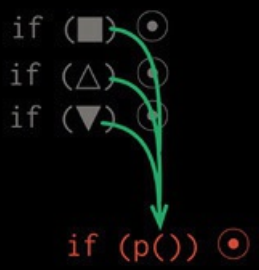
```
if (summer())
  charge = summerCharge();
```

```
else
  charge = regularCharge();

function summer() {
  return !aDate.isBefore(plan.summerStart) && !aDate.isAfter(plan.summerEnd);
}
function summerCharge() {
  return quantity * plan.summerRate;
}
function regularCharge() {
  return quantity * plan.regularRate + plan.regularServiceCharge;
}
```

Depois disso, gosto de reformatar a condicional usando o operador ternário.

```
charge = summer() ? summerCharge() : regularCharge();

function summer() {
  return !aDate.isBefore(plan.summerStart) && !aDate.isAfter(plan.summerEnd);
}
function summerCharge() {
  return quantity * plan.summerRate;
}
function regularCharge() {
  return quantity * plan.regularRate + plan.regularServiceCharge;
}
```

Consolidar expressão condicional (Consolidate Conditional Expression)




```
if (anEmployee.seniority < 2) return 0;
if (anEmployee.monthsDisabled > 12)
return 0;
if (anEmployee.isPartTime) return 0;
```

```
if (isNotEligableForDisability()) return 0;

function isNotEligableForDisability() {
  return ((anEmployee.seniority < 2)
          || (anEmployee.monthsDisabled >
12)
          || (anEmployee.isPartTime));
}
```

Motivação

Às vezes, deparo com uma série de verificações condicionais em que cada verificação é diferente, embora a ação resultante seja a mesma. Quando vejo isso, utilizo os operadores `and` e `or` para consolidá-las em uma única verificação condicional, com um único resultado.

Consolidar o código condicional é importante por dois motivos. Primeiro, deixa o código mais claro, mostrando que estou realmente fazendo uma única verificação que combina outras verificações. Usar uma sequência tem o mesmo efeito, porém passa a impressão de que estou executando uma sequência de verificações separadas que, somente por acaso, estão próximas. O segundo motivo pelo qual gosto de consolidar um código condicional é que, com frequência, isso me deixa preparado para usar *Extrair função (Extract Function)*. Extrair uma condição é uma das melhores tarefas que posso fazer para deixar meu código mais claro. Com isso, substituo uma instrução que mostra o que estou fazendo por outra que informa por que estou fazendo.

Os motivos a favor de consolidar condicionais também apontam para os motivos para não o fazer. Se eu considerar que as verificações são de fato independentes, e que não devem ser pensadas como uma única verificação, não devo fazer a refatoração.

Procedimento

- Garanta que nenhuma das condicionais tenha qualquer efeito colateral.

  Se alguma delas tiver, utilize *Separar consulta de modificador (Separate Query from Modifier)* antes.
- Tome duas das instruções condicionais e combine suas condições usando

um operador lógico.

Sequências são combinadas com `or`, e instruções `if` aninhadas são combinadas com `and`.

- Teste.
- Repita a operação de combinar condicionais até que todas estejam em uma única condição.
- Considere usar *Extrair função (Extract Function)* na condição resultante.

Exemplo

Observando atentamente um código, vejo o seguinte:

```
function disabilityAmount(anEmployee) {
  if (anEmployee.seniority < 2) return 0;
  if (anEmployee.monthsDisabled > 12) return 0;
  if (anEmployee.isPartTime) return 0;
  // calcula a aposentadoria por invalidez (disability amount)
```

É uma sequência de verificações condicionais, todas com o mesmo resultado. Como o resultado é o mesmo, essas condições devem ser combinadas em uma única expressão. Para uma sequência como essa, faço a combinação com um operador `or`.

```
function disabilityAmount(anEmployee) {
  if ((anEmployee.seniority < 2)
      || (anEmployee.monthsDisabled > 12)) return 0;
  if (anEmployee.isPartTime) return 0;
  // calcula a aposentadoria por invalidez (disability amount)
```

Testo, e depois incluo a outra condição:

```
function disabilityAmount(anEmployee) {
  if ((anEmployee.seniority < 2)
      || (anEmployee.monthsDisabled > 12)
      || (anEmployee.isPartTime)) return 0;
  // calcula a aposentadoria por invalidez (disability amount)
```

Depois que todas as condições estiverem reunidas, uso *Extrair função (Extract Function)*.

```
function disabilityAmount(anEmployee) {
  if (isNotEligableForDisability()) return 0;
  // calcula a aposentadoria por invalidez (disability amount)

  function isNotEligableForDisability() {
    return ((anEmployee.seniority < 2)
```

```
            || (anEmployee.monthsDisabled > 12)
            || (anEmployee.isPartTime));
    }
```

Exemplo: usando and

O exemplo anterior mostrou instruções combinadas com or, mas posso deparar com casos que exijam ands também. Um caso como esse apresenta instruções if aninhadas:

```
if (anEmployee.onVacation)
  if (anEmployee.seniority > 10)
    return 1;
return 0.5;
```

Combino essas condições usando um operador and.

```
if ((anEmployee.onVacation)
    && (anEmployee.seniority > 10)) return 1;
return 0.5;
```

Se eu tiver condições como essas misturadas, posso combiná-las usando operadores and e or conforme forem necessários. Se isso acontecer, é provável que o código fique confuso, portanto uso *Extrair função (Extract Function)* tranquilamente para deixar tudo mais compreensível.

Substituir condicional aninhada por cláusulas de guarda (Replace Nested Conditional with Guard Clauses)


```
function getPayAmount() {
  let result;
  if (isDead)
    result = deadAmount();
```

```
    else {
      if (isSeparated)
        result = separatedAmount();
      else {
        if (isRetired)
          result = retiredAmount();
        else
          result =
  normalPayAmount();
      }
    }
    return result;
  }
```

```
function getPayAmount() {
  if (isDead) return deadAmount();
  if (isSeparated) return
separatedAmount();
  if (isRetired) return retiredAmount();
  return normalPayAmount();
}
```

Motivação

De modo geral, vejo que as expressões condicionais se apresentam em dois estilos. No primeiro, os dois ramos da condicional fazem parte do comportamento normal, enquanto, no segundo, um ramo é normal e o outro indica uma condição incomum.

Esses tipos de condicionais têm propósitos diferentes – e esses propósitos devem estar evidentes no código. Se ambas as condições fizerem parte do comportamento normal, utilizo uma condição com ramos `if` e `else`. Se a condição for incomum, faço a sua verificação e retorno se ela for verdadeira. Esse tipo de verificação muitas vezes é chamado de **cláusula de guarda** (guard clause).

O ponto principal de *Substituir condicional aninhada por cláusulas de guarda* está na ênfase. Se eu usar uma construção if-then-else, estarei dando pesos iguais aos ramos `if` e `else`. Isso diz ao leitor que os ramos são igualmente prováveis e têm a mesma importância. Em contrapartida, a cláusula de guarda

diz que “essa não é a parte essencial desta função; se ela ocorrer, faça algo e saia”.

Em geral, percebo que uso *Substituir condicional aninhada por cláusulas de guarda* quando estou trabalhando com um programador que aprendeu que deve haver somente um único ponto de entrada e um único ponto de saída em um método. Ter um único ponto de entrada é garantido pelas linguagens modernas, mas ter um único ponto de saída não é uma regra realmente útil. A clareza é o princípio fundamental: se o método for mais claro tendo um único ponto de saída, use um ponto de saída; caso contrário, não faça isso.

Procedimento

- Selecione a condição mais externa a ser substituída e altere-a para uma cláusula de guarda.
- Teste.
- Repita conforme necessário.
- Se todas as cláusulas de guarda devolverem o mesmo resultado, use *Consolidar expressão condicional (Consolidate Conditional Expression)*.

Exemplo

A seguir, temos um código que calcula o valor de um pagamento (payment amount) para um funcionário (employee). Ele só será relevante se o funcionário ainda estiver na empresa, portanto deve haver verificações para os outros dois casos em que o funcionário não está.

```
function payAmount(employee) {
  let result;
  if(employee.isSeparated) {
    result = {amount: 0, reasonCode: "SEP"};
  }
  else {
    if (employee.isRetired) {
      result = {amount: 0, reasonCode: "RET"};
    }
    else {
      // lógica para calcular o valor
      lorem.ipsum(dolor.sitAmet);
      consectetur(adipiscing).elit();
      sed.do.eiusmod = tempor.incididunt.ut(labore) && dolore(magna.aliqua);
```

```
        ut.enim.ad(minim.veniam);
        result = someFinalComputation();
      }
    }
    return result;
  }
```

Aninhar as condicionais, nesse caso, mascara o verdadeiro sentido do que está acontecendo. O principal propósito desse código se aplicará somente se essas condições não se derem. Nessa situação, a intenção do código será mais facilmente percebida com cláusulas de guarda.

Como em qualquer mudança por refatoração, gosto de dar passos pequenos, portanto começo com a condição que aparece antes.

```
  function payAmount(employee) {
    let result;
    if (employee.isSeparated) return {amount: 0, reasonCode: "SEP"};
    if (employee.isRetired) {
      result = {amount: 0, reasonCode: "RET"};
    }
    else {
      // lógica para calcular o valor
      lorem.ipsum(dolor.sitAmet);
      consectetur(adipiscing).elit();
      sed.do.eiusmod = tempor.incididunt.ut(labore) && dolore(magna.aliqua);
      ut.enim.ad(minim.veniam);
      result = someFinalComputation();
    }
    return result;
  }
```

Testo essa mudança e passo para a próxima.

```
  function payAmount(employee) {
    let result;
    if (employee.isSeparated) return {amount: 0, reasonCode: "SEP"};
    if (employee.isRetired) return {amount: 0, reasonCode: "RET"};
    // lógica para calcular o valor
    lorem.ipsum(dolor.sitAmet);
    consectetur(adipiscing).elit();
    sed.do.eiusmod = tempor.incididunt.ut(labore) && dolore(magna.aliqua);
    ut.enim.ad(minim.veniam);
    result = someFinalComputation();
    return result;
  }
```

Nesse ponto, a variável de resultado não está realmente fazendo nada útil, portanto eu a removo.

```
function payAmount(employee) {
  let result;
  if (employee.isSeparated) return {amount: 0, reasonCode: "SEP"};
  if (employee.isRetired) return {amount: 0, reasonCode: "RET"};
  // lógica para calcular o valor
  lorem.ipsum(dolor.sitAmet);
  consectetur(adipiscing).elit();
  sed.do.eiusmod = tempor.incididunt.ut(labore) && dolore(magna.aliqua);
  ut.enim.ad(minim.veniam);
  return someFinalComputation();
}
```

A regra diz que você sempre ganhará um morango extra se remover uma variável mutável.

Exemplo: invertendo as condições

Ao revisar a versão preliminar da primeira edição deste livro, Joshua Kerievsky destacou que aplicamos *Substituir condicional aninhada por cláusulas de guarda* frequentemente invertendo as expressões condicionais. Melhor ainda, ele me deu um exemplo, e eu nem precisei exigir mais da minha imaginação.

```
function adjustedCapital(anInstrument) {
  let result = 0;
  if (anInstrument.capital > 0) {
    if (anInstrument.interestRate > 0 && anInstrument.duration > 0) {
      result = (anInstrument.income / anInstrument.duration) * anInstrument.adjustmentFactor;
    }
  }
  return result;
}
```

Novamente, faço as substituições, uma de cada vez; dessa vez, porém, inverto a condição quando coloco a cláusula de guarda.

```
function adjustedCapital(anInstrument) {
  let result = 0;
  if (anInstrument.capital <= 0) return result;
  if (anInstrument.interestRate > 0 && anInstrument.duration > 0) {
    result = (anInstrument.income / anInstrument.duration) * anInstrument.adjustmentFactor;
  }
  return result;
```

```
}
```

A próxima condicional é um pouco mais complicada, portanto faço a modificação em dois passos. Em primeiro lugar, simplesmente acrescento um not.

```
function adjustedCapital(anInstrument) {
  let result = 0;
  if (anInstrument.capital <= 0) return result;
  if (!(anInstrument.interestRate > 0 && anInstrument.duration > 0)) return result;
  result = (anInstrument.income / anInstrument.duration) * anInstrument.adjustmentFactor;
  return result;
}
```

Deixar nots em uma condicional como essa dá um nó realmente complicado em minha cabeça, portanto faço uma simplificação:

```
function adjustedCapital(anInstrument) {
  let result = 0;
  if (anInstrument.capital <= 0) return result;
  if (anInstrument.interestRate <= 0 || anInstrument.duration <= 0) return result;
  result = (anInstrument.income / anInstrument.duration) * anInstrument.adjustmentFactor;
  return result;
}
```

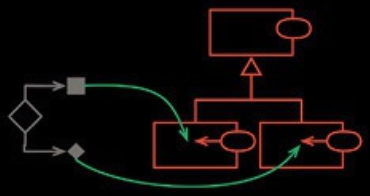
As duas linhas têm condições com o mesmo resultado, portanto aplico *Consolidar expressão condicional (Consolidate Conditional Expression)*.

```
function adjustedCapital(anInstrument) {
  let result = 0;
  if ( anInstrument.capital <= 0
       || anInstrument.interestRate <= 0
       || anInstrument.duration <= 0) return result;
  result = (anInstrument.income / anInstrument.duration) * anInstrument.adjustmentFactor;
  return result;
}
```

A variável result tem duas funções nesse exemplo. Sua primeira atribuição para zero indica o que deve ser devolvido quando a cláusula de guarda é acionada; seu segundo valor contém o cálculo final. Posso me livrar dela eliminando o seu uso duplo, o que me faz ganhar um morango.

```
function adjustedCapital(anInstrument) {
  if ( anInstrument.capital <= 0
       || anInstrument.interestRate <= 0
       || anInstrument.duration <= 0) return 0;
  return (anInstrument.income / anInstrument.duration) * anInstrument.adjustmentFactor;
}
```

Substituir condicional por polimorfismo (Replace Conditional with Polymorphism)



```
switch (bird.type) {
  case 'EuropeanSwallow':
    return "average";
  case 'AfricanSwallow':
    return (bird.numberOfCoconuts > 2) ? "tired" :
"average";
  case 'NorwegianBlueParrot':
    return (bird.voltage > 100) ? "scorched" : "beautiful";
  default:
    return "unknown";
```

```
class EuropeanSwallow {
  get plumage() {
    return "average";
  }
class AfricanSwallow {
  get plumage() {
      return (this.numberOfCoconuts > 2) ? "tired" :
"average";
  }
class NorwegianBlueParrot {
  get plumage() {
      return (this.voltage > 100) ? "scorched" : "beautiful";
  }
```

Motivação

Uma lógica condicional complexa é um dos elementos mais difíceis de entender em programação, portanto sempre procuro maneiras de acrescentar estrutura em uma lógica condicional. Com frequência, percebo que é possível

dividir a lógica em circunstâncias distintas – casos de alto nível – a fim de separar as condições. Às vezes é suficiente representar essa separação na estrutura da própria condicional, mas usar classes e polimorfismo pode deixar a divisão mais explícita.

Um caso comum é aquele em que é possível criar um conjunto de tipos, cada qual tratando a lógica condicional de modo distinto. Posso notar que livros, música e comida variam quanto ao modo como são tratados em razão de seu tipo. Isso se torna mais evidente quando há várias funções com uma instrução switch com base em um código de tipo. Nesse caso, removo a duplicação da lógica comum do switch criando classes para cada caso e usando polimorfismo para evidenciar o comportamento específico a cada tipo.

Outra situação é aquela em que posso pensar na lógica como um caso básico com variantes. O caso básico pode ser o mais comum ou o mais simples. Posso colocar essa lógica em uma superclasse, o que me permite pensar nela sem ter de me preocupar com as variantes. Então coloco cada caso variante em uma subclasse, que expresso com um código que enfatiza a sua diferença em relação ao caso base.

O polimorfismo é um dos recursos principais da programação orientada a objetos – e, como qualquer recurso conveniente, ele é suscetível a um uso exagerado. Já conheci pessoas que argumentam que todos os exemplos de lógica condicional deveriam ser substituídos por polimorfismo. Não concordo com essa visão. A maior parte de minhas lógicas condicionais faz uso de instruções condicionais básicas – if/else e switch/case. Porém, quando vejo uma lógica condicional complexa que pode ser melhorada conforme discutimos antes, considero o polimorfismo uma ferramenta eficaz.

Procedimento

- Se não houver classes para um comportamento polimórfico, crie-as, junto com uma função de factory que devolva a instância correta.
- Use a função de factory no código que faz a chamada.
- Mova a função condicional para a superclasse.

Se a lógica condicional não for uma função autocontida, use *Extrair função (Extract Function)* para deixá-la assim.

- Escolha uma das subclasses. Crie um método na subclasse que sobrescreva o método da instrução condicional. Copie o corpo desse ramo

da instrução condicional para o método da subclasse e faça a sua adequação.

- Repita o processo para cada ramo da condicional.
- Deixe um caso default para o método da superclasse. Ou, se a superclasse tiver de ser abstrata, declare esse método como abstrato ou lance um erro para mostrar que a responsabilidade deve ser de uma subclasse.

Exemplo

Meu amigo tem uma coleção de pássaros (birds) e quer saber a rapidez com que eles podem voar e como é a sua plumagem (plumage). Assim, temos alguns programas pequenos para determinar essas informações.

```
function plumages(birds) {
  return new Map(birds.map(b => [b.name, plumage(b)]));
}
function speeds(birds) {
  return new Map(birds.map(b => [b.name, airSpeedVelocity(b)]));
}
function plumage(bird) {
  switch (bird.type) {
  case 'EuropeanSwallow':
    return "average";
  case 'AfricanSwallow':
    return (bird.numberOfCoconuts > 2) ? "tired" : "average";
  case 'NorwegianBlueParrot':
    return (bird.voltage > 100) ? "scorched" : "beautiful";
  default:
    return "unknown";
  }
}
function airSpeedVelocity(bird) {
  switch (bird.type) {
  case 'EuropeanSwallow':
    return 35;
  case 'AfricanSwallow':
    return 40 - 2 * bird.numberOfCoconuts;
  case 'NorwegianBlueParrot':
    return (bird.isNailed) ? 0 : 10 + bird.voltage / 10;
  default:
    return null;
  }
}
```

Temos duas operações diferentes que variam conforme o tipo do pássaro, portanto faz sentido criar classes e usar polimorfismo para cada comportamento específico de um tipo.

Começo usando *Combinar funções em classe (Combine Functions into Class)* em airSpeedVelocity e em plumage.

```
function plumage(bird) {
  return new Bird(bird).plumage;
}
function airSpeedVelocity(bird) {
  return new Bird(bird).airSpeedVelocity;
}
class Bird {
  constructor(birdObject) {
    Object.assign(this, birdObject);
  }
  get plumage() {
    switch (this.type) {
    case 'EuropeanSwallow':
      return "average";
    case 'AfricanSwallow':
      return (this.numberOfCoconuts > 2) ? "tired" : "average";
    case 'NorwegianBlueParrot':
      return (this.voltage > 100) ? "scorched" : "beautiful";
    default:
      return "unknown";
    }
  }
  get airSpeedVelocity() {
    switch (this.type) {
    case 'EuropeanSwallow':
      return 35;
    case 'AfricanSwallow':
      return 40 - 2 * this.numberOfCoconuts;
    case 'NorwegianBlueParrot':
      return (this.isNailed) ? 0 : 10 + this.voltage / 10;
    default:
      return null;
    }
  }
}
```

Agora adiciono subclasses para cada tipo de pássaro, junto com uma função de factory para instanciar a subclasse apropriada.

```
function plumage(bird) {
  return createBird(bird).plumage;
}
function airSpeedVelocity(bird) {
  return createBird(bird).airSpeedVelocity;
}
function createBird(bird) {
  switch (bird.type) {
  case 'EuropeanSwallow':
    return new EuropeanSwallow(bird);
  case 'AfricanSwallow':
    return new AfricanSwallow(bird);
  case 'NorweigianBlueParrot':
    return new NorwegianBlueParrot(bird);
  default:
    return new Bird(bird);
  }
}
class EuropeanSwallow extends Bird {
}

class AfricanSwallow extends Bird {
}

class NorwegianBlueParrot extends Bird {
}
```

Agora que já criei a estrutura de classes necessária, posso começar a trabalhar nos dois métodos condicionais. Começarei com a plumagem. Escolho um ramo da instrução switch e sobrescrevo-o na subclasse apropriada.

class EuropeanSwallow…

```
get plumage() {
  return "average";
}
```

class Bird…

```
get plumage() {
  switch (this.type) {
  case 'EuropeanSwallow':
    throw "oops";
  case 'AfricanSwallow':
    return (this.numberOfCoconuts > 2) ? "tired" : "average";
  case 'NorwegianBlueParrot':
    return (this.voltage > 100) ? "scorched" : "beautiful";
```

```
    default:
      return "unknown";
    }
  }
```

Coloco o throw porque sou paranoico.

Posso compilar e testar nesse ponto. Em seguida, se tudo estiver bem, considero o próximo ramo.

class AfricanSwallow…

```
  get plumage() {
    return (this.numberOfCoconuts > 2) ? "tired" : "average";
  }
```

Depois, cuido do Norwegian Blue:

class NorwegianBlueParrot…

```
  get plumage() {
    return (this.voltage > 100) ? "scorched" : "beautiful";
  }
```

Deixo o método da superclasse para o caso default.

class Bird…

```
  get plumage() {
    return "unknown";
  }
```

Repito o mesmo processo para airSpeedVelocity. Depois que terminar, terei o código a seguir (também internalizei as funções de nível mais alto para airSpeedVelocity e plumage):

```
  function plumages(birds) {
    return new Map(birds
                   .map(b => createBird(b))
                   .map(bird => [bird.name, bird.plumage]));
  }
  function speeds(birds) {
    return new Map(birds
                   .map(b => createBird(b))
                   .map(bird => [bird.name, bird.airSpeedVelocity]));
  }
  function createBird(bird) {
    switch (bird.type) {
    case 'EuropeanSwallow':
      return new EuropeanSwallow(bird);
```

```
    case 'AfricanSwallow':
      return new AfricanSwallow(bird);
    case 'NorwegianBlueParrot':
      return new NorwegianBlueParrot(bird);
    default:
      return new Bird(bird);
    }
}
class Bird {
    constructor(birdObject) {
      Object.assign(this, birdObject);
    }
    get plumage() {
      return "unknown";
    }
    get airSpeedVelocity() {
      return null;
    }
}
class EuropeanSwallow extends Bird {
    get plumage() {
      return "average";
    }
    get airSpeedVelocity() {
      return 35;
    }
}

class AfricanSwallow extends Bird {
    get plumage() {
      return (this.numberOfCoconuts > 2) ? "tired" : "average";
    }
    get airSpeedVelocity() {
      return 40 - 2 * this.numberOfCoconuts;
    }
}
class NorwegianBlueParrot extends Bird {
    get plumage() {
      return (this.voltage > 100) ? "scorched" : "beautiful";
    }
    get airSpeedVelocity() {
      return (this.isNailed) ? 0 : 10 + this.voltage / 10;
    }
}
```

Observando esse código final, vejo que a superclasse Bird não é estritamente necessária. Em JavaScript, não é preciso ter uma hierarquia de tipos para usar o polimorfismo; desde que meus objetos implementem os métodos devidamente nomeados, tudo funcionará sem problemas. Nessa situação, porém, prefiro manter a superclasse desnecessária, pois ela ajuda a explicar o modo como as classes estão relacionadas no domínio.

Exemplo: usando polimorfismo para variação

No exemplo com os pássaros, estou usando claramente uma hierarquia de generalização. É assim que subclasses e polimorfismo em geral são discutidos em livros didáticos (inclusive no meu) – mas essa não é a única maneira como a herança é usada na prática; com efeito, provavelmente não é a mais comum nem a melhor maneira. Outro caso de herança ocorre quando quero indicar que um objeto é semelhante a outro em sua maior parte, mas apresenta algumas variações.

Como exemplo desse caso, considere um código usado por uma agência de classificação (rating agency) para calcular uma taxa de investimentos para viagens de navio. A agência de classificação atribui uma classificação “A” ou “B”, dependendo de diversos fatores associados a riscos e potencial para lucro. O risco é calculado com base na avaliação da natureza da viagem, assim como no histórico das viagens anteriores do capitão.

```
function rating(voyage, history) {
  const vpf = voyageProfitFactor(voyage, history);
  const vr = voyageRisk(voyage);
  const chr = captainHistoryRisk(voyage, history);
  if (vpf * 3 > (vr + chr * 2)) return "A";
  else return "B";
}

function voyageRisk(voyage) {
  let result = 1;
  if (voyage.length > 4) result += 2;
  if (voyage.length > 8) result += voyage.length - 8;
  if (["china", "east-indies"].includes(voyage.zone)) result += 4;
  return Math.max(result, 0);
}
function captainHistoryRisk(voyage, history) {
  let result = 1;
  if (history.length < 5) result += 4;
  result += history.filter(v => v.profit < 0).length;
```

```
  if (voyage.zone === "china" && hasChina(history)) result -= 2;
  return Math.max(result, 0);
}
function hasChina(history) {
  return history.some(v => "china" === v.zone);
}
function voyageProfitFactor(voyage, history) {
  let result = 2;
  if (voyage.zone === "china") result += 1;
  if (voyage.zone === "east-indies") result += 1;
  if (voyage.zone === "china" && hasChina(history)) {
    result += 3;
    if (history.length > 10) result += 1;
    if (voyage.length > 12) result += 1;
    if (voyage.length > 18) result -= 1;
  }
  else {
    if (history.length > 8) result += 1;
    if (voyage.length > 14) result -= 1;
  }
  return result;
}
```

As funções `voyageRisk` e `captainHistoryRisk` atribuem pontos para o risco, `voyageProfitFactor` atribui pontos para o lucro em potencial e `rating` os combina a fim de apresentar a classificação geral da viagem.

O código que faz a chamada tem o seguinte aspecto:

```
const voyage = {zone: "west-indies", length: 10};
const history = [
  {zone: "east-indies", profit: 5},
  {zone: "west-indies", profit: 15},
  {zone: "china",       profit: -2},
  {zone: "west-africa", profit: 7},
];

const myRating = rating(voyage, history);
```

O que quero enfatizar, nesse caso, é o fato de haver dois lugares que usam a mesma lógica condicional para tratar o caso de uma viagem para a China cujo capitão já esteve antes nesse país.

```
function rating(voyage, history) {
  const vpf = voyageProfitFactor(voyage, history);
  const vr = voyageRisk(voyage);
  const chr = captainHistoryRisk(voyage, history);
```

```
  if (vpf * 3 > (vr + chr * 2)) return "A";
  else return "B";
}

function voyageRisk(voyage) {
  let result = 1;
  if (voyage.length > 4) result += 2;
  if (voyage.length > 8) result += voyage.length - 8;
  if (["china", "east-indies"].includes(voyage.zone)) result += 4;
  return Math.max(result, 0);
}
function captainHistoryRisk(voyage, history) {
  let result = 1;
  if (history.length < 5) result += 4;
  result += history.filter(v => v.profit < 0).length;
  if (voyage.zone === "china" && hasChina(history)) result -= 2;
  return Math.max(result, 0);
}
function hasChina(history) {
  return history.some(v => "china" === v.zone);
}
function voyageProfitFactor(voyage, history) {
  let result = 2;
  if (voyage.zone === "china") result += 1;
  if (voyage.zone === "east-indies") result += 1;
  if (voyage.zone === "china" && hasChina(history)) {
    result += 3;
    if (history.length > 10) result += 1;
    if (voyage.length > 12) result += 1;
    if (voyage.length > 18) result -= 1;
  }
  else {
    if (history.length > 8) result += 1;
    if (voyage.length > 14) result -= 1;
  }
  return result;
}
```

Usarei herança e polimorfismo para separar a lógica que trata esses casos da lógica básica. Essa é uma refatoração particularmente útil se eu estiver prestes a introduzir outras lógicas especiais para esse caso – e a lógica para essas viagens repetidas à China puderem dificultar a compreensão do caso básico.

Começo com um conjunto de funções. Para introduzir o polimorfismo, é

necessário criar uma estrutura de classes, portanto começo aplicando *Combinar funções em classe (Combine Functions into Class)*. Isso resulta no código a seguir:

```
function rating(voyage, history) {
  return new Rating(voyage, history).value;
}
class Rating {
  constructor(voyage, history) {
    this.voyage = voyage;
    this.history = history;
  }
  get value() {
    const vpf = this.voyageProfitFactor;
    const vr = this.voyageRisk;
    const chr = this.captainHistoryRisk;
    if (vpf * 3 > (vr + chr * 2)) return "A";
    else return "B";
  }
  get voyageRisk() {
    let result = 1;
    if (this.voyage.length > 4) result += 2;
    if (this.voyage.length > 8) result += this.voyage.length - 8;
    if (["china", "east-indies"].includes(this.voyage.zone)) result += 4;
    return Math.max(result, 0);
  }
  get captainHistoryRisk() {
    let result = 1;
    if (this.history.length < 5) result += 4;
    result += this.history.filter(v => v.profit < 0).length;
    if (this.voyage.zone === "china" && this.hasChinaHistory) result -= 2;
    return Math.max(result, 0);
  }
  get voyageProfitFactor() {
    let result = 2;

    if (this.voyage.zone === "china") result += 1;
    if (this.voyage.zone === "east-indies") result += 1;
    if (this.voyage.zone === "china" && this.hasChinaHistory) {
      result += 3;
      if (this.history.length > 10) result += 1;
      if (this.voyage.length > 12) result += 1;
      if (this.voyage.length > 18) result -= 1;
    }
```

```
    else {
      if (this.history.length > 8) result += 1;
      if (this.voyage.length > 14) result -= 1;
    }
    return result;
  }

  get hasChinaHistory() {
    return this.history.some(v => "china" === v.zone);
  }
}
```

Com isso, tenho a classe para o caso de base. Agora preciso criar uma subclasse vazia que contenha o comportamento variante.

```
class ExperiencedChinaRating extends Rating {
}
```

Em seguida, crio uma função de factory para devolver a classe variante quando necessário.

```
function createRating(voyage, history) {
  if (voyage.zone === "china" && history.some(v => "china" === v.zone))
    return new ExperiencedChinaRating(voyage, history);
  else return new Rating(voyage, history);
}
```

É preciso modificar qualquer chamada para que a função de factory seja usada em vez de chamar diretamente o construtor; nesse caso, é somente a função de classificação.

```
function rating(voyage, history) {
  return createRating(voyage, history).value;
}
```

Há dois comportamentos que devem ser movidos para uma subclasse. Começo com a lógica em `captainHistoryRisk`:

class Rating…

```
get captainHistoryRisk() {
  let result = 1;
  if (this.history.length < 5) result += 4;
  result += this.history.filter(v => v.profit < 0).length;
  if (this.voyage.zone === "china" && this.hasChinaHistory) result -= 2;
  return Math.max(result, 0);
}
```

Escrevo o método que sobrescreve o código na subclasse:

class ExperiencedChinaRating

```
get captainHistoryRisk() {
  const result = super.captainHistoryRisk - 2;
  return Math.max(result, 0);
}
```

class Rating…

```
get captainHistoryRisk() {
  let result = 1;
  if (this.history.length < 5) result += 4;
  result += this.history.filter(v => v.profit < 0).length;
  if (this.voyage.zone === "china" && this.hasChinaHistory) result -= 2;
  return Math.max(result, 0);
}
```

Separar o comportamento variante de voyageProfitFactor é um pouco mais confuso. Não posso simplesmente remover o comportamento variante e chamar o método da superclasse, pois há um caminho alternativo nesse caso. Também não quero copiar o método inteiro da superclasse para a subclasse.

class Rating…

```
get voyageProfitFactor() {
  let result = 2;

  if (this.voyage.zone === "china") result += 1;
  if (this.voyage.zone === "east-indies") result += 1;
  if (this.voyage.zone === "china" && this.hasChinaHistory) {
    result += 3;
    if (this.history.length > 10) result += 1;
    if (this.voyage.length > 12) result += 1;
    if (this.voyage.length > 18) result -= 1;
  }
  else {
    if (this.history.length > 8) result += 1;
    if (this.voyage.length > 14) result -= 1;
  }
  return result;
}
```

Então minha resposta é usar *Extrair função (Extract Function)* antes em todo o bloco condicional.

class Rating…

```
get voyageProfitFactor() {
  let result = 2;
```

```
    if (this.voyage.zone === "china") result += 1;
    if (this.voyage.zone === "east-indies") result += 1;
    result += this.voyageAndHistoryLengthFactor;
    return result;
  }

  get voyageAndHistoryLengthFactor() {
    let result = 0;
    if (this.voyage.zone === "china" && this.hasChinaHistory) {
      result += 3;
      if (this.history.length > 10) result += 1;
      if (this.voyage.length > 12) result += 1;
      if (this.voyage.length > 18) result -= 1;
    }
    else {
      if (this.history.length > 8) result += 1;
      if (this.voyage.length > 14) result -= 1;
    }
    return result;
  }
```

Um nome de função com um “And” exala muito mau cheiro, mas vou deixá-la como está por um tempo enquanto crio a subclasse.

class Rating…

```
  get voyageAndHistoryLengthFactor() {
    let result = 0;
    if (this.history.length > 8) result += 1;
    if (this.voyage.length > 14) result -= 1;
    return result;
  }
```

class ExperiencedChinaRating…

```
  get voyageAndHistoryLengthFactor() {
    let result = 0;
    result += 3;
    if (this.history.length > 10) result += 1;
    if (this.voyage.length > 12) result += 1;
    if (this.voyage.length > 18) result -= 1;
    return result;
  }
```

Formalmente, é o fim da refatoração – separei o comportamento variante na subclasse. A lógica da superclasse está mais simples de compreender e é mais fácil trabalhar com ela; só terei de lidar com o caso variante quando estiver

trabalhando com o código da subclasse, o qual é expresso em termos de sua diferença em relação à superclasse.

No entanto, sinto que devo pelo menos descrever o que eu faria com o novo método desajeitado. Introduzir um método exclusivamente para ser sobrescrito por uma subclasse é comum quando usamos esse tipo de herança com caso de base e variação. No entanto, um método grosseiro como esse encobre o que está acontecendo, em vez de revelar.

O "And" denuncia que há, na verdade, duas modificações separadas ocorrendo nesse caso – portanto, acho que as separar seria uma atitude inteligente. Farei isso usando *Extrair função (Extract Function)* na modificação relacionada ao tamanho do histórico, tanto na superclasse quanto na subclasse. Começarei somente com a superclasse:

class Rating…

```
get voyageAndHistoryLengthFactor() {
  let result = 0;
  result += this.historyLengthFactor;
  if (this.voyage.length > 14) result -= 1;
  return result;
}
get historyLengthFactor() {
  return (this.history.length > 8) ? 1 : 0;
}
```

Faço o mesmo com a subclasse:

class ExperiencedChinaRating…

```
get voyageAndHistoryLengthFactor() {
  let result = 0;
  result += 3;
  result += this.historyLengthFactor;
  if (this.voyage.length > 12) result += 1;
  if (this.voyage.length > 18) result -= 1;
  return result;
}
get historyLengthFactor() {
  return (this.history.length > 10) ? 1 : 0;
}
```

Posso então usar *Mover instruções para os pontos de chamada (Move Statements to Callers)* no caso da superclasse.

class Rating…

```
  get voyageProfitFactor() {
    let result = 2;
    if (this.voyage.zone === "china") result += 1;
    if (this.voyage.zone === "east-indies") result += 1;
    result += this.historyLengthFactor;
    result += this.voyageAndHistoryLengthFactor;
    return result;
  }
  get voyageAndHistoryLengthFactor() {
    let result = 0;
    result += this.historyLengthFactor;
    if (this.voyage.length > 14) result -= 1;
    return result;
  }
```

class ExperiencedChinaRating…

```
  get voyageAndHistoryLengthFactor() {
    let result = 0;
    result += 3;
    result += this.historyLengthFactor;
    if (this.voyage.length > 12) result += 1;
    if (this.voyage.length > 18) result -= 1;
    return result;
  }
```

Em seguida, eu usaria *Renomear função (Rename Function)*.

class Rating…

```
  get voyageProfitFactor() {
    let result = 2;
    if (this.voyage.zone === "china") result += 1;
    if (this.voyage.zone === "east-indies") result += 1;
    result += this.historyLengthFactor;
    result += this.voyageLengthFactor;
    return result;
  }
  get voyageLengthFactor() {
    return (this.voyage.length > 14) ? - 1: 0;
  }
```

Mudo para um operador ternário a fim de simplificar voyageLengthFactor.

class ExperiencedChinaRating…

```
  get voyageLengthFactor() {
    let result = 0;
```

```
    result += 3;
    if (this.voyage.length > 12) result += 1;
    if (this.voyage.length > 18) result -= 1;
    return result;
  }
```

Um último detalhe. Não acho que somar 3 pontos faça sentido como parte do fator duração da viagem (voyage length factor) – é melhor somá-lo no resultado geral.

class ExperiencedChinaRating…

```
  get voyageProfitFactor() {
    return super.voyageProfitFactor + 3;
  }
  get voyageLengthFactor() {
    let result = 0;
    result += 3;
    if (this.voyage.length > 12) result += 1;
    if (this.voyage.length > 18) result -= 1;
    return result;
  }
```

No final da refatoração, terei o código a seguir. Inicialmente há uma classe básica de classificação que pode ignorar qualquer complicação para o caso da China:

```
  class Rating {
    constructor(voyage, history) {
      this.voyage = voyage;
      this.history = history;
    }
    get value() {
      const vpf = this.voyageProfitFactor;
      const vr = this.voyageRisk;
      const chr = this.captainHistoryRisk;
      if (vpf * 3 > (vr + chr * 2)) return "A";
      else return "B";
    }
    get voyageRisk() {
      let result = 1;
      if (this.voyage.length > 4) result += 2;
      if (this.voyage.length > 8) result += this.voyage.length - 8;
      if (["china", "east-indies"].includes(this.voyage.zone)) result += 4;
      return Math.max(result, 0);
    }
```

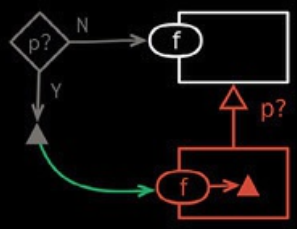
```
  get captainHistoryRisk() {
    let result = 1;
    if (this.history.length < 5) result += 4;
    result += this.history.filter(v => v.profit < 0).length;
    return Math.max(result, 0);
  }
  get voyageProfitFactor() {
    let result = 2;
    if (this.voyage.zone === "china") result += 1;
    if (this.voyage.zone === "east-indies") result += 1;
    result += this.historyLengthFactor;
    result += this.voyageLengthFactor;
    return result;
  }
  get voyageLengthFactor() {
    return (this.voyage.length > 14) ? - 1: 0;
  }
  get historyLengthFactor() {
    return (this.history.length > 8) ? 1 : 0;
  }
}
```

O código para o caso de China é lido como um conjunto de variações em relação à base:

```
class ExperiencedChinaRating extends Rating {
  get captainHistoryRisk() {
    const result = super.captainHistoryRisk - 2;
    return Math.max(result, 0);
  }
  get voyageLengthFactor() {
    let result = 0;
    if (this.voyage.length > 12) result += 1;
    if (this.voyage.length > 18) result -= 1;
    return result;
  }
  get historyLengthFactor() {
    return (this.history.length > 10) ? 1 : 0;
  }
  get voyageProfitFactor() {
    return super.voyageProfitFactor + 3;
  }
}
```

Introduzir caso especial (Introduce Special Case)

anteriormente: *Introduzir objeto nulo* (Introduce Null Object)




```
if (aCustomer === "unknown") customerName =
"occupant";
```

```
class UnknownCustomer {
    get name() {return
"occupant";}
```

Motivação

Um caso comum de código duplicado ocorre quando muitos usuários de uma estrutura de dados verificam um valor específico, e então a maioria deles faz o mesmo. Se eu encontrar muitas partes na base de código que apresentem a mesma reação a um valor em particular, gostaria de levar essa reação para um único lugar.

Um bom procedimento para isso é usar o padrão Caso Especial (Special Case), no qual crio um elemento para o caso especial que capture todo o comportamento comum. Isso permite que eu substitua a maior parte das verificações de casos especiais por chamadas simples.

Um caso especial pode se manifestar de diversas maneiras. Se tudo que eu estiver fazendo com o objeto é ler dados, posso fornecer um objeto literal com todos os valores necessários preenchidos. Se eu precisar de outros comportamentos além de valores simples, posso criar um objeto especial, com métodos para todos os comportamentos comuns. O objeto de caso especial pode ser devolvido por uma classe que faça um encapsulamento, ou pode ser inserido em uma estrutura de dados com uma transformação.

Um valor comum que exige um processamento de caso especial é o valor nulo (null), e é por isso que esse padrão muitas vezes é chamado de padrão de Objeto Nulo (Null Object). Porém, a abordagem é a mesma para qualquer caso especial – gosto de dizer que Objeto Nulo (Null Object) é um caso especial de Caso Especial (Special Case).

Procedimento

Comece com uma estrutura de dados contêiner (ou classe) que tenha uma propriedade, a qual será o assunto (subject) da refatoração. Clientes do contêiner comparam a propriedade subject do contêiner com um valor de caso especial. Queremos substituir o valor de caso especial referente ao assunto por uma classe de caso especial ou por uma estrutura de dados.

- Adicione uma propriedade para verificação de caso especial no assunto, devolvendo falso.
- Crie um objeto para o caso especial somente com a propriedade para verificação do caso especial, devolvendo verdadeiro.
- Aplique *Extrair função (Extract Function)* no código de comparação do caso especial. Garanta que todos os clientes usem a nova função em vez de fazer diretamente a comparação.
- Introduza o novo caso especial no código, seja devolvendo-o com uma chamada de função ou aplicando uma função de transformação.
- Modifique o corpo da função de comparação do caso especial de modo que ele utilize a propriedade de verificação do caso especial.
- Teste.
- Use *Combinar funções em classe (Combine Functions into Class)* ou *Combinar funções em transformação (Combine Functions into Transform)* para mover todo o comportamento comum do caso especial para o novo elemento.

Como a classe do caso especial geralmente devolve valores fixos para requisições simples, estas podem ser tratadas fazendo com que o caso especial seja um registro literal.

- Utilize *Internalizar função (Inline Function)* na função de comparação do caso especial nos lugares em que ainda for necessário.

Exemplo

Uma empresa de serviços públicos instala seus serviços em certos endereços (sites).

class Site…

```
get customer() {return this._customer;}
```

Há diversas propriedades na classe cliente (customer); considerarei três delas.

class Customer…

```
get name() {...}
get billingPlan() {...}
set billingPlan(arg) {...}
get paymentHistory() {...}
```

Na maioria das vezes, um endereço tem um cliente; às vezes, porém, não haverá nenhum. Alguém pode ter saído, e eu ainda não sei quem – se houver alguém – passou a ocupar o endereço. Quando isso acontece, o registro de dados tem o campo de cliente preenchido com a string “unknown” (desconhecido). Como essa situação pode acontecer, os clientes da classe endereço devem ser capazes de lidar com um cliente desconhecido. Eis alguns fragmentos de exemplo:

cliente 1…

```
const aCustomer = site.customer;
// ... muito código intermediário ...
let customerName;
if (aCustomer === "unknown") customerName = "occupant";
else customerName = aCustomer.name;
```

cliente 2…

```
const plan = (aCustomer === "unknown") ?
        registry.billingPlans.basic
        : aCustomer.billingPlan;
```

cliente 3…

```
if (aCustomer !== "unknown") aCustomer.billingPlan = newPlan;
```

cliente 4…

```
const weeksDelinquent = (aCustomer === "unknown") ?
        0
        : aCustomer.paymentHistory.weeksDelinquentInLastYear;
```

Observando a base de código, vejo que muitos clientes do objeto endereço

(site) devem lidar com um cliente desconhecido. A maioria deles faz o mesmo quando depara com um cliente desse tipo: usa “occupant” (ocupante) como o nome, atribui um plano básico de tarifação a eles e os classifica como devedores por zero semana (zero-weeks delinquent). São esses testes espalhados para um caso especial, além de uma resposta comum, que me dizem que é hora de usar um Objeto de Caso Especial (Special Case Object).

Começo acrescentando um método na classe cliente para indicar se ele é desconhecido.

class Customer…

```
get isUnknown() {return false;}
```

Então acrescento uma classe para Cliente Desconhecido (Unknown Customer).

```
class UnknownCustomer {
  get isUnknown() {return true;}
}
```

Observe que não faço de `UnknownCustomer` uma subclasse de `Customer`. Em outras linguagens, em particular naquelas que são estaticamente tipadas, eu teria feito isso, mas as regras de JavaScript para subclasses bem como a sua tipagem dinâmica fazem com que, nesse caso, seja melhor não fazer.

Agora temos a parte complicada. Tenho de devolver esse novo objeto de caso especial sempre que `"unknown"` é esperado, e mudar cada teste que verifique se o valor é esse para que passe a usar o novo método `isUnknown`. Em geral, sempre gosto de organizar o código para que seja possível fazer uma pequena alteração de cada vez e então testar. No entanto, se eu modificar a classe cliente para que devolva um cliente desconhecido no lugar de “unknown”, tenho de fazer com que todo cliente que teste se é “unknown” chame `isUnknown` – e tenho de fazer tudo de uma só vez. Acho isso tão tentador quanto comer fígado (ou seja, nem um pouco).

Há uma técnica comum que pode ser usada sempre que me vejo diante desse problema. Uso *Extrair função (Extract Function)* no código que devo alterar em vários lugares – nesse caso, no código de comparação com o caso especial.

```
function isUnknown(arg) {
  if (!((arg instanceof Customer) || (arg === "unknown")))
    throw new Error(`investigate bad value: <${arg}>`);
  return (arg === "unknown");
```

```
}
```

Coloquei uma armadilha nesse código no caso de um valor inesperado. Isso pode me ajudar a identificar qualquer erro ou comportamento estranho enquanto faço a refatoração.

Agora posso usar essa função sempre que estiver testando se um cliente é desconhecido. Posso alterar essas chamadas, uma por vez, e testar após cada mudança.

cliente 1…

```
let customerName;
if (isUnknown(aCustomer)) customerName = "occupant";
else customerName = aCustomer.name;
```

Depois de um tempo, terei feito tudo.

cliente 2…

```
const plan = (isUnknown(aCustomer)) ?
        registry.billingPlans.basic
        : aCustomer.billingPlan;
```

cliente 3…

```
if (!isUnknown(aCustomer)) aCustomer.billingPlan = newPlan;
```

cliente 4…

```
const weeksDelinquent = isUnknown(aCustomer) ?
        0
        : aCustomer.paymentHistory.weeksDelinquentInLastYear;
```

Depois que eu tiver alterado todas as chamadas para que usem `isUnknown`, poderei modificar a classe de endereço para que devolva um cliente desconhecido.

class Site…

```
get customer() {
  return (this._customer === "unknown") ? new UnknownCustomer() : this._customer;
}
```

Posso verificar se não estou mais usando a string “unknown” modificando `isUnknown` para que use o valor para cliente desconhecido.

cliente 1…

```
function isUnknown(arg) {
  if (!(arg instanceof Customer || arg instanceof UnknownCustomer))
    throw new Error(`investigate bad value: <${arg}>`);
  return arg.isUnknown;
```

```
}
```

Testo para garantir que tudo esteja funcionando.

Agora começa a diversão. Posso usar *Combinar funções em classe (Combine Functions into Class)* para observar cada verificação de caso especial dos clientes e ver se posso fazer uma substituição por um valor comumente esperado. No momento, tenho vários clientes usando “occupant” para o nome de um cliente desconhecido, assim:

cliente 1…

```
let customerName;
if (isUnknown(aCustomer)) customerName = "occupant";
else customerName = aCustomer.name;
```

Acrescento um método apropriado no cliente desconhecido:

class UnknownCustomer…

```
get name() {return "occupant";}
```

Agora posso remover todo esse código condicional.

cliente 1…

```
const customerName = aCustomer.name;
```

Depois de ter testado e verificado que esse código funciona, provavelmente poderei usar *Internalizar variável (Inline Variable)* nessa variável também.

A seguir, temos a propriedade para o plano de tarifação (billing plan).

cliente 2…

```
const plan = (isUnknown(aCustomer)) ?
        registry.billingPlans.basic
        : aCustomer.billingPlan;
```

cliente 3…

```
if (!isUnknown(aCustomer)) aCustomer.billingPlan = newPlan;
```

Para o comportamento de leitura, faço o mesmo que fiz com o nome – devolvo a resposta comum. Para o comportamento de escrita, o código atual não chama o setter para um cliente desconhecido – assim, para o caso especial, deixo o setter ser chamado, porém ele não faz nada.

class UnknownCustomer…

```
get billingPlan() {return registry.billingPlans.basic;}
set billingPlan(arg) { /* ignora */ }
```

cliente que lê…

```
const plan = aCustomer.billingPlan;
```

cliente que escreve…

```
aCustomer.billingPlan = newPlan;
```

Objetos de casos especiais são objetos de valor e, desse modo, devem ser sempre imutáveis, mesmo que os objetos que eles estejam substituindo não sejam.

O último caso é um pouco mais complexo porque o caso especial deve devolver outro objeto que tem propriedades próprias.

cliente…

```
const weeksDelinquent = isUnknown(aCustomer) ?
    0
    : aCustomer.paymentHistory.weeksDelinquentInLastYear;
```

De acordo com a regra geral para um objeto de caso especial, se ele tiver de devolver objetos relacionados, em geral eles serão, por si só, casos especiais. Nesse exemplo, preciso criar um histórico de pagamentos nulo (null payment history).

class UnknownCustomer…

```
get paymentHistory() {return new NullPaymentHistory();}
```

class NullPaymentHistory…

```
get weeksDelinquentInLastYear() {return 0;}
```

cliente…

```
const weeksDelinquent = aCustomer.paymentHistory.weeksDelinquentInLastYear;
```

Prossigo observando todos os clientes para ver se posso substituí-los pelo comportamento polimórfico. Contudo, haverá exceções – clientes que queiram fazer algo diferente com o caso especial. Posso ter 23 clientes que usem “occupant” para o nome de um cliente desconhecido, mas sempre haverá um que necessitará de algo diferente.

cliente…

```
const name = ! isUnknown(aCustomer) ? aCustomer.name : "unknown occupant";
```

Nesse caso, devo manter uma verificação de caso especial. Vou alterá-lo para que use o método do cliente, essencialmente usando *Internalizar função (Inline Function)* em isUnknown.

cliente…

```
const name = aCustomer.isUnknown ? "unknown occupant" : aCustomer.name;
```

Depois de ter modificado todos os clientes, poderei usar *Remover código morto (Remove Dead Code)* na função global isUnknown, pois não deverá haver mais ninguém chamando essa função.

Exemplo: usando um objeto literal

Criar uma classe como essa exige certo trabalho, considerando que se trata apenas de um valor simples. Porém, no exemplo que apresentei, tive de criar a classe, pois o cliente poderia ser atualizado. Entretanto, se eu somente lesse a estrutura de dados, seria possível usar um objeto literal.

Eis o caso inicial de novo – é o mesmo, exceto que, dessa vez, não há nenhum cliente fazendo uma atualização:

class Site…

```
get customer() {return this._customer;}
```

class Customer…

```
get name() {...}
get billingPlan() {...}
set billingPlan(arg) {...}
get paymentHistory() {...}
```

cliente 1…

```
const aCustomer = site.customer;
// ... muito código intermediário ...
let customerName;
if (aCustomer === "unknown") customerName = "occupant";
else customerName = aCustomer.name;
```

cliente 2…

```
const plan = (aCustomer === "unknown") ?
        registry.billingPlans.basic
        : aCustomer.billingPlan;
```

cliente 3…

```
const weeksDelinquent = (aCustomer === "unknown") ?
        0
        : aCustomer.paymentHistory.weeksDelinquentInLastYear;
```

Como no caso anterior, começo acrescentando uma propriedade isUnknown no cliente e criando um objeto de caso especial com esse campo. A diferença é que, dessa vez, o caso especial é um literal.

class Customer…

```
  get isUnknown() {return false;}
```

nível mais alto…

```
  function createUnknownCustomer() {
    return {
      isUnknown: true,
    };
  }
```

Aplico *Extrair função (Extract Function)* no teste da condição de caso especial.

```
  function isUnknown(arg) {
    return (arg === "unknown");
  }
```

cliente 1…

```
  let customerName;
  if (isUnknown(aCustomer)) customerName = "occupant";
  else customerName = aCustomer.name;
```

cliente 2…

```
  const plan = isUnknown(aCustomer) ?
          registry.billingPlans.basic
          : aCustomer.billingPlan;
```

cliente 3…

```
  const weeksDelinquent = isUnknown(aCustomer) ?
          0
          : aCustomer.paymentHistory.weeksDelinquentInLastYear;
```

Modifico a classe de endereço (site) e o teste de condição para que funcionem com o caso especial.

class Site…

```
  get customer() {
    return (this._customer === "unknown") ? createUnknownCustomer() : this._customer;
  }
```

nível mais alto…

```
  function isUnknown(arg) {
    return arg.isUnknown;
  }
```

Então substituo cada resposta padrão pelo valor literal apropriado. Começo pelo nome:

```
function createUnknownCustomer() {
  return {
    isUnknown: true,
    name: "occupant",
  };
}
```

cliente 1…

```
const customerName = aCustomer.name;
```

En seguida, cuido do plano de tarifação:

```
function createUnknownCustomer() {
  return {
    isUnknown: true,
    name: "occupant",
    billingPlan: registry.billingPlans.basic,
  };
}
```

cliente 2…

```
const plan = aCustomer.billingPlan;
```

De modo semelhante, posso criar um histórico de pagamento nulo aninhado com o literal:

```
function createUnknownCustomer() {
  return {
    isUnknown: true,
    name: "occupant",
    billingPlan: registry.billingPlans.basic,
    paymentHistory: {
      weeksDelinquentInLastYear: 0,
    },
  };
}
```

cliente 3…

```
const weeksDelinquent = aCustomer.paymentHistory.weeksDelinquentInLastYear;
```

Se eu usar um literal como esse, devo deixá-lo imutável, o que pode ser feito com `freeze`. Em geral, prefiro usar uma classe.

Exemplo: usando uma transformação

Os dois casos anteriores envolvem uma classe, mas a mesma ideia pode ser aplicada a um registro usando um passo de transformação.

Vamos supor que nossa entrada seja uma estrutura de registro simples como o código a seguir:

```
{
  name: "Acme Boston",
  location: "Malden MA",
  // outros detalhes do endereço
  customer: {
    name: "Acme Industries",
    billingPlan: "plan-451",
    paymentHistory: {
      weeksDelinquentInLastYear: 7
      // outras informações
    },
    // outras informações
  }
}
```

Em alguns casos, o cliente não é conhecido, e casos como esses são marcados do mesmo modo:

```
{
  name: "Warehouse Unit 15",
  location: "Malden MA",
  // outros detalhes do endereço
  customer: "unknown",
}
```

Tenho um código de cliente parecido, que verifica se o cliente é desconhecido:

cliente 1…

```
const site = acquireSiteData();
const aCustomer = site.customer;
// ... muito código intermediário ...
let customerName;
if (aCustomer === "unknown") customerName = "occupant";
else customerName = aCustomer.name;
```

cliente 2…

```
const plan = (aCustomer === "unknown") ?
        registry.billingPlans.basic
        : aCustomer.billingPlan;
```

cliente 3…

```
const weeksDelinquent = (aCustomer === "unknown") ?
```

```
        0
        : aCustomer.paymentHistory.weeksDelinquentInLastYear;
```

Meu primeiro passo é submeter a estrutura de dados do endereço a uma transformação que, no momento, não faz nada além de uma cópia profunda (deep copy).

cliente 1…

```
const rawSite = acquireSiteData();
const site = enrichSite(rawSite);
const aCustomer = site.customer;
// ... muito código intermediário ...
let customerName;
if (aCustomer === "unknown") customerName = "occupant";
else customerName = aCustomer.name;
function enrichSite(inputSite) {
  return _.cloneDeep(inputSite);
}
```

Aplico *Extrair função (Extract Function)* no teste de cliente desconhecido.

```
function isUnknown(aCustomer) {
  return aCustomer === "unknown";
}
```

cliente 1…

```
const rawSite = acquireSiteData();
const site = enrichSite(rawSite);
const aCustomer = site.customer;
// ... muito código intermediário ...
let customerName;
if (isUnknown(aCustomer)) customerName = "occupant";
else customerName = aCustomer.name;
```

cliente 2…

```
const plan = (isUnknown(aCustomer)) ?
        registry.billingPlans.basic
        : aCustomer.billingPlan;
```

cliente 3…

```
const weeksDelinquent = (isUnknown(aCustomer)) ?
        0
        : aCustomer.paymentHistory.weeksDelinquentInLastYear;
```

Inicio o enriquecimento acrescentando uma propriedade isUnknown no cliente.

```
function enrichSite(aSite) {
```

```
  const result = _.cloneDeep(aSite);
  const unknownCustomer = {
    isUnknown: true,
  };

  if (isUnknown(result.customer)) result.customer = unknownCustomer;
  else result.customer.isUnknown = false;
  return result;
}
```

Posso então modificar o teste da condição de caso especial de modo a incluir uma sondagem dessa nova propriedade. Mantenho o teste original também para que o teste funcione tanto para os endereços puros quanto para os endereços enriquecidos.

```
function isUnknown(aCustomer) {
  if (aCustomer === "unknown") return true;
  else return aCustomer.isUnknown;
}
```

Testo para garantir que tudo está bem e, então, começo aplicando *Combinar funções em transformação (Combine Functions into Transform)* no caso especial. Inicialmente movo a opção de nome para a função de enriquecimento.

```
function enrichSite(aSite) {
  const result = _.cloneDeep(aSite);
  const unknownCustomer = {
    isUnknown: true,
    name: "occupant",
  };

  if (isUnknown(result.customer)) result.customer = unknownCustomer;
  else result.customer.isUnknown = false;
  return result;
}
```

cliente 1…

```
const rawSite = acquireSiteData();
const site = enrichSite(rawSite);
const aCustomer = site.customer;
// ... muito código intermediário ...
const customerName = aCustomer.name;
```

Testo, e então cuido do plano de tarifação.

```
function enrichSite(aSite) {
```

```
  const result = _.cloneDeep(aSite);
  const unknownCustomer = {
    isUnknown: true,
    name: "occupant",
    billingPlan: registry.billingPlans.basic,
  };

  if (isUnknown(result.customer)) result.customer = unknownCustomer;
  else result.customer.isUnknown = false;
  return result;
}
```

cliente 2…

```
  const plan = aCustomer.billingPlan;
```

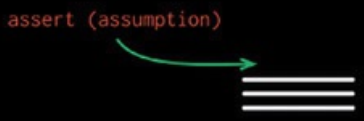
Testo novamente, e então trato o último código cliente.

```
function enrichSite(aSite) {
  const result = _.cloneDeep(aSite);
  const unknownCustomer = {
    isUnknown: true,
    name: "occupant",
    billingPlan: registry.billingPlans.basic,
    paymentHistory: {
      weeksDelinquentInLastYear: 0,
    }
  };

  if (isUnknown(result.customer)) result.customer = unknownCustomer;
  else result.customer.isUnknown = false;
  return result;
}
```

cliente 3…

```
const weeksDelinquent = aCustomer.paymentHistory.weeksDelinquentInLastYear;
```

Introduzir asserção (Introduce Assertion)




```
if (this.discountRate)
  base = base - (this.discountRate *
base);
```

```
assert(this.discountRate >= 0);
if (this.discountRate)
  base = base - (this.discountRate *
base);
```

Motivação

Com frequência, seções de código funcionam somente se determinadas condições forem verdadeiras. Isso pode ser tão simples quanto um cálculo de raiz quadrada funcionar apenas para um valor de entrada positivo. Com um objeto, pode-se exigir que pelo menos um campo de um grupo de campos contenha um valor.

Tais pressupostos muitas vezes não são explícitos e só poderão ser deduzidos analisando um algoritmo. Às vezes, os pressupostos são explicitados com um comentário. Uma técnica melhor consiste em deixar o pressuposto explícito escrevendo uma asserção.

Uma asserção é uma instrução condicional que se supõe que seja sempre verdadeira. Uma falha em uma asserção sinaliza um erro do programador. Falhas de asserção jamais devem ser verificadas por outras partes do sistema. As asserções devem ser escritas de modo que o programa funcione igualmente de modo correto, mesmo que todas sejam removidas; com efeito, algumas linguagens oferecem asserções que podem ser desativadas com uma flag de compilação.

Muitas vezes, vejo pessoas incentivarem o uso de asserções a fim de encontrar erros. Embora, sem dúvida, seja algo bom, esse não é o único motivo para usá-las. Acho que as asserções são uma forma importante de comunicação – elas dizem algo ao leitor sobre o estado em que se supõe que esteja o programa no respectivo ponto de execução. Também as acho convenientes para depuração, e, em razão de sua importância para a comunicação, tendo a deixá-las no código depois de ter corrigido o erro que estava procurando. Um código autotestável reduz a importância das asserções na depuração, pois testes de unidade cada vez mais localizados em geral fazem um trabalho melhor, mas continuo gostando das asserções como instrumento de comunicação.

Procedimento

- Quando você notar que é suposto que uma condição seja verdadeira, adicione uma asserção para confirmá-la.

Como as asserções não devem afetar a execução de um sistema, o acréscimo de uma asserção sempre preserva o comportamento.

Exemplo

Eis um cálculo simples de descontos. Um cliente pode receber um desconto a ser aplicado em todas as suas compras:

class Customer…

```
applyDiscount(aNumber) {
  return (this.discountRate)
    ? aNumber - (this.discountRate * aNumber)
    : aNumber;
}
```

Há um pressuposto, nesse caso, de que a taxa de desconto é um número positivo. Posso deixar esse pressuposto explícito usando uma asserção. No entanto, não é possível inserir facilmente uma asserção em uma expressão ternária; assim, em primeiro lugar, reformularei o operador usando uma instrução if-then.

class Customer…

```
applyDiscount(aNumber) {
  if (!this.discountRate) return aNumber;
  else return aNumber - (this.discountRate * aNumber);
}
```

Agora posso acrescentar facilmente a asserção.

class Customer…

```
applyDiscount(aNumber) {
  if (!this.discountRate) return aNumber;
  else {
    assert(this.discountRate >= 0);
    return aNumber - (this.discountRate * aNumber);
  }
}
```

Nesse caso, eu preferiria colocar a asserção no método de escrita. Se a asserção falhar em `applyDiscount`, meu primeiro problema a ser resolvido seria como o valor foi parar nesse campo.

class Customer…

```
set discountRate(aNumber) {
  assert(null === aNumber || aNumber >= 0);
  this._discountRate = aNumber;
}
```

Uma asserção como essa pode ser particularmente importante se a origem do erro for difícil de identificar – pode ser um sinal negativo errado em um dado de entrada ou uma inversão em outro ponto do código.

Há um verdadeiro perigo no uso excessivo de asserções. Não uso asserções para verificar tudo que acho que seja verdadeiro, mas somente para verificar itens que *devam* ser verdadeiros. A duplicação é um problema em particular, pois é comum que sejam feitos ajustes nesses tipos de condições. Desse modo, acho essencial remover qualquer duplicação nessas condições, geralmente por meio de um uso generalizado de *Extrair função (Extract Function)*.

Uso asserções apenas para códigos que mostrem erros do programador. Se eu estiver lendo dados de origem externa, qualquer verificação de valor deve ser um código de primeira classe do programa, e não uma asserção – a menos que eu tenha de fato confiança na fonte externa. As asserções são o último recurso para ajudar a rastrear bugs – embora, ironicamente, eu as use apenas quando acho que elas jamais falharão.

CAPÍTULO 11
Refatorando APIs

Módulos e suas funções são os blocos de construção de nosso software. As APIs são as junções que usamos para conectá-los. Deixar essas APIs fáceis de entender e de usar é importante, mas também difícil: é necessário refatorá-las à medida que aprendo a aperfeiçoá-las.

Uma boa API separa claramente qualquer função que atualize dados daquelas que apenas os leem. Se eu vir essas funções combinadas, uso *Separar consulta de modificador (Separate Query from Modifier)* para separá-las. Posso unificar funções que variem somente por causa de um valor com *Parametrizar função (Parameterize Function)*. Alguns parâmetros, porém, são de fato apenas um sinal de um comportamento totalmente diferente, e podem ser extirpados de modo mais apropriado com *Remover argumento de flag (Remove Flag Argument)*.

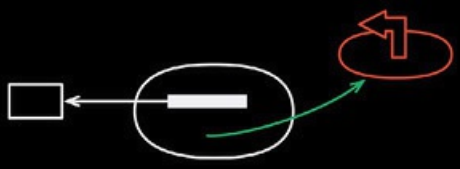
Estruturas de dados muitas vezes são desnecessariamente desempacotadas quando passadas entre funções; prefiro mantê-las juntas usando *Preservar objeto inteiro (Preserve Whole Object)*. Decisões sobre o que deve ser passado como parâmetro e o que pode ser resolvido pela função chamada devem ser frequentemente revistas com *Substituir parâmetro por consulta (Replace Parameter with Query)* e *Substituir consulta por parâmetro (Replace Query with Parameter)*.

Uma classe é uma forma comum de módulo. Prefiro que meus objetos sejam o máximo possível imutáveis, portanto uso *Remover método de escrita (Remove Setting Method)* sempre que puder. Com frequência, quando uma chamada pede um novo objeto, preciso de mais flexibilidade do que um construtor simples é capaz de oferecer; posso tê-la usando *Substituir construtor por função de factory (Replace Constructor with Factory Function)*.

As duas últimas refatorações tratam da dificuldade de separar uma função particularmente complexa que passe muitos dados. Posso transformar essa

função em um objeto com *Substituir função por comando (Replace Function with Command)*, facilitando usar *Extrair função (Extract Function)* no corpo da função. Se eu simplificar mais tarde a função e ela não for mais necessária como um objeto de comando, faço com que esse objeto volte a ser uma função com *Substituir comando por função (Replace Command with Function)*.

Separar consulta de modificador (Separate Query from Modifier)



```
function getTotalOutstandingAndSendBill() {
  const result = customer.invoices.reduce((total, each) => each.amount +
total, 0);
  sendBill();
  return result;
}
```

```
function totalOutstanding() {
  return customer.invoices.reduce((total, each) => each.amount +
total, 0);
}
function sendBill() {
  emailGateway.send(formatBill(customer));
}
```

Motivação

Quando tenho uma função que me forneça um valor e não gere efeitos colaterais observáveis, tenho algo muito valioso. Posso chamar essa função com a frequência que eu quiser; posso mover a chamada dessa função para outros lugares dentro de uma função e é mais fácil de testar. Em suma, tenho muito menos com que me preocupar.

É uma boa ideia sinalizar claramente a diferença entre funções com efeitos colaterais e funções sem eles. Uma boa regra a ser seguida determina que qualquer função que devolva um valor não deve ter efeitos colaterais observáveis – é a separação entre comando-consulta (command-query separation) [mf-cqs]. Alguns programadores tratam essa regra como absoluta. Não defendo uma pureza de 100% quanto a isso (assim como para tudo mais), mas tento segui-la na maioria das vezes, e ela tem me servido bem.

Se eu deparar com um método que devolva um valor, mas que também tenha efeitos colaterais, sempre tento separar a consulta do modificador.

Note que uso a expressão efeitos colaterais *observáveis*. Uma otimização comum é fazer cache do valor de uma consulta em um campo para agilizar as chamadas repetidas. Embora isso altere o estado do objeto com o cache, a mudança não é observável. Qualquer sequência de consultas sempre devolverá o mesmo resultado para cada consulta.

Procedimento

- Copie a função e nomeie-a como uma consulta.

Observe o corpo da função para ver o que é devolvido. Se a consulta for usada para preencher uma variável, o nome da variável deverá oferecer uma boa pista.

- Elimine qualquer efeito colateral da nova função de consulta.
- Execute verificações estáticas.
- Localize cada chamada do método original. Se uma chamada usar o valor de retorno, substitua a chamada original por uma chamada à consulta e insira uma chamada para o método original em seguida. Teste após cada mudança.
- Remova os valores de retorno do método original.
- Teste.

Com frequência, depois desse procedimento, haverá duplicação a ser removida entre a consulta e o método original.

Exemplo

Eis uma função que verifica uma lista de nomes em busca de um vilão (miscreant). Se um deles for encontrado, a função devolverá o nome desse vilão e disparará os alarmes. Isso é feito apenas para o primeiro vilão

encontrado (acho que um é suficiente).

```
function alertForMiscreant (people) {
  for (const p of people) {
    if (p === "Don") {
      setOffAlarms();
      return "Don";
    }
    if (p === "John") {
      setOffAlarms();
      return "John";
    }
  }
  return "";
}
```

Começo copiando a função, dando-lhe um nome baseado na qualidade de consulta da função.

```
function findMiscreant (people) {
  for (const p of people) {
    if (p === "Don") {
      setOffAlarms();
      return "Don";
    }
    if (p === "John") {
      setOffAlarms();
      return "John";
    }
  }
  return "";
}
```

Removo os efeitos colaterais dessa nova consulta.

```
function findMiscreant (people) {
  for (const p of people) {
    if (p === "Don") {
      ~~setOffAlarms();~~
      return "Don";
    }
    if (p === "John") {
      ~~setOffAlarms();~~
      return "John";
    }
  }
  return "";
```

```
}
```

Agora acesso cada chamada e substituo por uma chamada à consulta, seguida de uma chamada ao modificador. Assim,

```
const found = alertForMiscreant(people);
```

passa a ser:

```
const found = findMiscreant(people);
alertForMiscreant(people);
```

Então removo os valores de retorno do modificador.

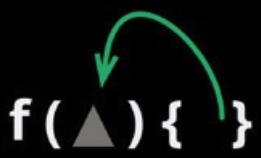
```
function alertForMiscreant (people) {
  for (const p of people) {
    if (p === "Don") {
      setOffAlarms();
      return;
    }
    if (p === "John") {
      setOffAlarms();
      return;
    }
  }
  return;
}
```

Agora tenho muitas duplicações entre o modificador original e a nova consulta, portanto posso usar *Substituir algoritmo (Substitute Algorithm)* para que o modificador utilize a consulta.

```
function alertForMiscreant (people) {
  if (findMiscreant(people) !== "") setOffAlarms();
}
```

Parametrizar função (Parameterize Function)

anteriormente: *Parametrizar método (Parameterize Method)*




```
function tenPercentRaise(aPerson) {
  aPerson.salary =
aPerson.salary.multiply(1.1);
```

```
}
function fivePercentRaise(aPerson) {
  aPerson.salary =
aPerson.salary.multiply(1.05);
}
```

```
function raise(aPerson, factor) {
  aPerson.salary = aPerson.salary.multiply(1 +
factor);
}
```

Motivação

Se vejo duas funções executando uma lógica muito parecida com valores literais distintos, posso remover a duplicação usando uma única função com parâmetros para os diferentes valores. Com isso, a função se torna mais útil, pois poderá ser aplicada em outros lugares com valores diferentes.

Procedimento

- Selecione um dos métodos similares.
- Use *Mudar declaração de função (Change Function Declaration)* para acrescentar quaisquer literais que devam se transformar em parâmetros.
- Em cada chamada da função, acrescente o valor literal.
- Teste.
- Modifique o corpo da função de modo que os novos parâmetros sejam usados. Teste após cada mudança.
- Para cada função similar, substitua a chamada por uma chamada à função parametrizada. Teste após cada mudança.

Se a função parametrizada original não funcionar para uma função similar, adapte-a para a nova função antes de passar para a próxima.

Exemplo

Um exemplo claro seria um código como este:

```
function tenPercentRaise(aPerson) {
  aPerson.salary = aPerson.salary.multiply(1.1);
```

```
}
function fivePercentRaise(aPerson) {
  aPerson.salary = aPerson.salary.multiply(1.05);
}
```

Felizmente é óbvio que é possível substituir esse código por:

```
function raise(aPerson, factor) {
  aPerson.salary = aPerson.salary.multiply(1 + factor);
}
```

No entanto, a situação pode ser um pouco mais complicada. Considere o código a seguir:

```
function baseCharge(usage) {
  if (usage < 0) return usd(0);
  const amount =
        bottomBand(usage) * 0.03
        + middleBand(usage) * 0.05
        + topBand(usage) * 0.07;
  return usd(amount);
}
function bottomBand(usage) {
  return Math.min(usage, 100);
}
function middleBand(usage) {
  return usage > 100 ? Math.min(usage, 200) - 100 : 0;
}
function topBand(usage) {
  return usage > 200 ? usage - 200 : 0;
}
```

Nesse caso, a lógica é claramente semelhante – porém, será semelhante o suficiente para que um método parametrizado seja criado para as bandas? Sim, é, mas talvez seja um pouco menos óbvio que no caso trivial anterior.

Quando tento parametrizar algumas funções relacionadas, minha abordagem consiste em tomar uma das funções e acrescentar-lhe parâmetros, com um olho nos demais casos. Com códigos para intervalos como esse, geralmente o ponto de partida deve ser o intervalo intermediário. Assim, trabalharei com `middleBand` de modo a modificá-la para que use parâmetros e, em seguida, farei as adaptações nas outras chamadas para que se adequem.

`middleBand` usa dois valores literais: `100` e `200`. Eles representam o início e o fim dessa banda intermediária. Começo usando *Mudar declaração de função (Change Function Declaration)* para acrescentar esses valores à chamada. Durante o processo, mudarei também o nome da função para algo que faça

sentido com a parametrização.

```
function withinBand(usage, bottom, top) {
  return usage > 100 ? Math.min(usage, 200) - 100 : 0;
}
function baseCharge(usage) {
  if (usage < 0) return usd(0);
  const amount =
        bottomBand(usage) * 0.03
        + withinBand(usage, 100, 200) * 0.05
        + topBand(usage) * 0.07;
  return usd(amount);
}
```

Substituo cada literal por uma referência ao parâmetro:

```
function withinBand(usage, bottom, top) {
  return usage > bottom ? Math.min(usage, 200) - bottom : 0;
}
```

e depois:

```
function withinBand(usage, bottom, top) {
  return usage > bottom ? Math.min(usage, top) - bottom : 0;
}
```

Substituo a chamada da função para a banda inferior por uma chamada à nova função parametrizada.

```
function baseCharge(usage) {
  if (usage < 0) return usd(0);
  const amount =
        withinBand(usage, 0, 100) * 0.03
        + withinBand(usage, 100, 200) * 0.05
        + topBand(usage) * 0.07;
  return usd(amount);
}
function bottomBand(usage) {
  return Math.min(usage, 100);
}
```

Para substituir a chamada de função para a banda superior, é necessário fazer uso de infinito.

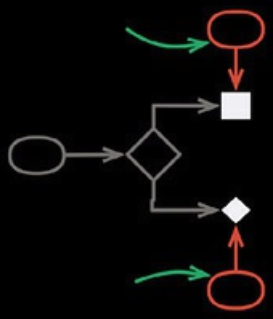
```
function baseCharge(usage) {
  if (usage < 0) return usd(0);
  const amount =
        withinBand(usage, 0, 100) * 0.03
        + withinBand(usage, 100, 200) * 0.05
```

```
        + withinBand(usage, 200, Infinity) * 0.07;
    return usd(amount);
  }
  function topBand(usage) {
    return usage > 200 ? usage - 200 : 0;
  }
```

Com a lógica funcionando do modo como está agora, eu poderia remover a cláusula de guarda inicial. No entanto, embora seja desnecessária do ponto de vista lógico, gostaria de manter essa cláusula, pois ela documenta o modo de tratar esse caso.

Remover argumento de flag (Remove Flag Argument)

anteriormente: *Substituir parâmetro por métodos explícitos* (Replace Parameter with Explicit Methods)



```
function setDimension(name,
value) {
  if (name === "height") {
    this._height = value;
    return;
  }
  if (name === "width") {
    this._width = value;
    return;
  }
}
```

```
function setHeight(value) {this._height =
```

```
value;}
function setWidth (value) {this._width =
value;}
```

Motivação

Um argumento de flag é um argumento utilizado por quem chama uma função a fim de indicar a lógica que a função chamada deve executar. Posso chamar uma função como esta:

```
function bookConcert(aCustomer, isPremium) {
  if (isPremium) {
    // lógica para reserva premium
  } else {
    // lógica para reserva comum
  }
}
```

Para reservar (book) um concerto premium, faço a chamada assim:

```
bookConcert(aCustomer, true);
```

Argumentos de flag também podem estar na forma de enumerados:

```
bookConcert(aCustomer, CustomerType.PREMIUM);
```

ou como strings (ou símbolos, em linguagens que os usem):

```
bookConcert(aCustomer, "premium");
```

Não gosto de argumentos de flag porque eles complicam o processo para entender quais são as chamadas de função disponíveis e como chamá-las. Em minha primeira incursão por uma API, em geral analiso a lista de funções disponíveis, e os argumentos de flag ocultam as diferenças nas chamadas dessas funções. Após selecionar uma função, é necessário descobrir quais são os valores disponíveis para os argumentos de flag. Flags booleanas são piores ainda, pois elas não informam seu significado para o leitor – em uma chamada de função, não é possível descobrir o que `true` significa. Disponibilizar uma função explícita para a tarefa que quero executar é uma opção mais clara.

```
premiumBookConcert(aCustomer);
```

Nem todos os argumentos como esses são argumentos de flag. Para ser um argumento de flag, quem faz as chamadas deve definir o valor booleano com um valor literal, e não com dados que fluam pelo programa. Além disso, a função de implementação deve usar o argumento para influenciar o seu controle de fluxo, e não como um dado a ser passado para outras funções.

Remover argumentos de flag não só deixa o código mais claro como também auxilia minhas ferramentas. Ferramentas de análise de código agora poderão ver mais facilmente a diferença entre chamar a lógica premium e chamar a lógica comum.

Argumentos de flag podem ter sua utilidade caso haja mais de um deles na função, pois, do contrário, seria necessário ter funções explícitas para cada combinação de seus valores. Contudo, isso também é um sinal de que uma função está fazendo tarefas demais, e eu deveria procurar uma maneira de criar funções mais simples, que possam ser compostas para implementar essa lógica.

Procedimento

- Crie uma função explícita para cada valor do parâmetro.
- Se a função principal tiver uma condicional clara para disparar a lógica, use *Decompor condicional (Decompose Conditional)* para criar as funções explícitas. Caso contrário, crie funções que façam um encapsulamento.
- Para cada chamada que utilize um valor literal para o parâmetro, faça uma substituição por uma chamada à função explícita.

Exemplo

Observando um código, vejo chamadas para calcular uma data de entrega (delivery date) de uma remessa (shipment). Algumas chamadas têm o seguinte aspecto:

```
aShipment.deliveryDate = deliveryDate(anOrder, true);
```

enquanto outras são assim:

```
aShipment.deliveryDate = deliveryDate(anOrder, false);
```

Diante de um código como esse, começo imediatamente a me perguntar sobre o significado do valor booleano. O que ele faz?

Eis o código do corpo de `deliveryDate`:

```
function deliveryDate(anOrder, isRush) {
  if (isRush) {
    let deliveryTime;
    if (["MA", "CT"] .includes(anOrder.deliveryState)) deliveryTime = 1;
    else if (["NY", "NH"].includes(anOrder.deliveryState)) deliveryTime = 2;
    else deliveryTime = 3;
    return anOrder.placedOn.plusDays(1 + deliveryTime);
```

```
    }
    else {
      let deliveryTime;
      if (["MA", "CT", "NY"].includes(anOrder.deliveryState)) deliveryTime = 2;
      else if (["ME", "NH"] .includes(anOrder.deliveryState)) deliveryTime = 3;
      else deliveryTime = 4;
      return anOrder.placedOn.plusDays(2 + deliveryTime);
    }
  }
```

Nesse caso, a chamada usa um valor booleano literal para determinar qual código deve ser executado – um argumento de flag clássico. Porém, o ponto principal ao usar uma função é seguir as instruções de quem chama, portanto é melhor deixar mais clara a intenção de quem faz a chamada usando funções explícitas.

Nesse exemplo, posso fazer isso usando *Decompor condicional (Decompose Conditional)*, que me dá o seguinte resultado:

```
function deliveryDate(anOrder, isRush) {
  if (isRush) return rushDeliveryDate(anOrder);
  else return regularDeliveryDate(anOrder);
}
function rushDeliveryDate(anOrder) {
    let deliveryTime;
    if (["MA", "CT"] .includes(anOrder.deliveryState)) deliveryTime = 1;
    else if (["NY", "NH"].includes(anOrder.deliveryState)) deliveryTime = 2;
    else deliveryTime = 3;
    return anOrder.placedOn.plusDays(1 + deliveryTime);
}
function regularDeliveryDate(anOrder) {
    let deliveryTime;
    if (["MA", "CT", "NY"].includes(anOrder.deliveryState)) deliveryTime = 2;
    else if (["ME", "NH"] .includes(anOrder.deliveryState)) deliveryTime = 3;
    else deliveryTime = 4;
    return anOrder.placedOn.plusDays(2 + deliveryTime);
}
```

As duas novas funções capturam melhor o propósito da chamada, portanto posso substituir cada chamada de

```
aShipment.deliveryDate = deliveryDate(anOrder, true);
```

por

```
aShipment.deliveryDate = rushDeliveryDate(anOrder);
```

e fazer algo semelhante com o outro caso.

Depois que tiver substituído todas as chamadas, removo deliveryDate.

Um argumento de flag não se constitui somente pela presença de um valor booleano – é fato de o booleano ser definido com um literal em vez de ser preenchido com um dado. Se todas as chamadas de deliveryDate fossem do tipo:

```
const isRush = determineIfRush(anOrder);
aShipment.deliveryDate = deliveryDate(anOrder, isRush);
```

eu não teria nenhum problema com a assinatura de deliveryDate (apesar disso, eu ainda aplicaria *Decompor condicional (Decompose Conditional))*.

Talvez algumas chamadas usem o argumento como uma flag definindo-o com um literal, enquanto outras definem o argumento com um dado. Nesse caso, eu ainda usaria *Remover argumento de flag*, mas não mudaria as chamadas com dados e não removeria deliveryDate no final. Assim, ofereço suporte para as duas interfaces, com seus usos distintos.

Decompor a condicional dessa forma é um bom modo de conduzir essa refatoração, mas só funcionará se o disparo da lógica, feito com base no parâmetro, for a parte externa da função (ou se eu puder refatorá-la facilmente para que isso ocorra). Também é possível que o parâmetro seja usado de modo muito mais complexo, como nesta versão alternativa de deliveryDate:

```
function deliveryDate(anOrder, isRush) {
  let result;
  let deliveryTime;
  if (anOrder.deliveryState === "MA" || anOrder.deliveryState === "CT")
    deliveryTime = isRush? 1 : 2;
  else if (anOrder.deliveryState === "NY" || anOrder.deliveryState === "NH") {
    deliveryTime = 2;
    if (anOrder.deliveryState === "NH" && !isRush)
      deliveryTime = 3;
  }
  else if (isRush)
    deliveryTime = 3;
  else if (anOrder.deliveryState === "ME")
    deliveryTime = 3;
  else
    deliveryTime = 4;
  result = anOrder.placedOn.plusDays(2 + deliveryTime);
  if (isRush) result = result.minusDays(1);
  return result;
}
```

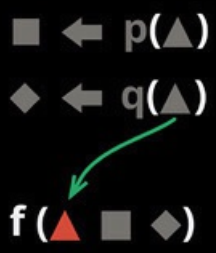
Nesse caso, levar isRush para uma condicional que dispare a lógica no nível mais alto provavelmente será mais trabalhoso do que eu gostaria. Assim, como alternativa, posso criar uma camada de funções sobre deliveryDate:

```
function rushDeliveryDate (anOrder) {return deliveryDate(anOrder, true);}
function regularDeliveryDate(anOrder) {return deliveryDate(anOrder, false);}
```

Essas funções de encapsulamento são essencialmente aplicações parciais de deliveryDate, embora sejam definidas no texto do programa em vez de o serem por composição de funções.

Posso então fazer a mesma substituição das chamadas, conforme fiz antes com a condicional decomposta. Se não houver nenhuma chamada que use o parâmetro como um dado, prefiro restringir a visibilidade da função ou renomeá-la de modo a sinalizar que ela não deve ser usada diretamente (por exemplo, deliveryDateHelperOnly).

Preservar objeto inteiro (Preserve Whole Object)




```
const low =
aRoom.daysTempRange.low;
const high =
aRoom.daysTempRange.high;
if (aPlan.withinRange(low, high))
```

```
if
(aPlan.withinRange(aRoom.daysTempRange))
```

Motivação

Se eu vir um código que faça a derivação de alguns valores de um registro e então passe esses valores para uma função, gostaria de substituí-los pelo

próprio registro completo, deixando o corpo da função fazer a derivação dos valores necessários.

Passar o registro completo é melhor para lidar com alterações caso a função chamada necessite de mais dados desse registro no futuro – essa alteração não exigirá que eu altere a lista de parâmetros. Com isso, o tamanho da lista de parâmetros também será reduzido, em geral deixando a chamada da função mais fácil de entender. Se muitas funções forem chamadas com os dados parciais, com frequência elas duplicarão a lógica que manipula esses dados – uma lógica que em geral poderá ser movida para o objeto completo.

O principal motivo para não fazer essa refatoração seria não querer que a função chamada estabeleça uma dependência com o objeto completo – ocorrerá tipicamente se eles estiverem em módulos distintos.

Extrair diversos valores de um objeto somente para executar uma lógica neles é um mau cheiro no código (veja a seção *Inveja de recursos* no Capítulo 3, *página* 101), e em geral é um sinal de que essa lógica deve ser movida para o próprio objeto completo. *Preservar objeto inteiro* é particularmente comum após ter usado *Introduzir objeto de parâmetros (Introduce Parameter Object)*, quando busco quaisquer ocorrências do conjunto de dados originais a fim de substituí-los pelo novo objeto.

Se várias porções de código usarem somente o mesmo subconjunto dos recursos de um objeto, isso pode sinalizar uma boa oportunidade para o uso de *Extrair classe (Extract Class)*.

Um caso que muitas pessoas podem não perceber é aquele em que um objeto chama outro com vários de seus próprios dados. Se vejo isso, posso substituir esses valores por uma autorreferência (`this` em JavaScript).

Procedimento

- Crie uma função vazia com os parâmetros desejados.
- Dê um nome à função que seja fácil de pesquisar, para que ele seja substituído no final.
- Preencha o corpo da nova função com uma chamada à função antiga, fazendo um mapeamento dos novos parâmetros para os parâmetros antigos.
- Execute verificações estáticas.
- Adapte cada chamada de modo a usar a nova função, testando após cada

mudança.

- Isso pode significar que um código que faça a derivação do parâmetro não seja mais necessário, portanto poderá estar sujeito a *Remover código morto (Remove Dead Code)*.
- Depois que todas as chamadas originais tiverem sido modificadas, use *Internalizar função (Inline Function)* na função original.
- Modifique o nome da nova função e todas as suas chamadas.

Exemplo

Considere um sistema de monitoração de quartos. Ele compara sua faixa diária de temperaturas com uma faixa em um plano de aquecimento (heating plan) predefinido.

chamada...

```
const low = aRoom.daysTempRange.low;
const high = aRoom.daysTempRange.high;
if (!aPlan.withinRange(low, high))
  alerts.push("room temperature went outside range");
```

class HeatingPlan…

```
withinRange(bottom, top) {
  return (bottom >= this._temperatureRange.low) && (top <= this._temperatureRange.high);
}
```

Em vez de desempacotar a informação de faixa ao passá-la, posso passar o objeto de faixa completo.

Começo definindo a interface que quero ter como uma função vazia.

class HeatingPlan…

```
xxNEWwithinRange(aNumberRange) {
}
```

Como minha intenção é que essa função substitua o `withinRange` existente, eu lhe atribuo o mesmo nome, mas com um prefixo facilmente substituível.

Então acrescento o corpo da função, que chama o `withinRange` existente. Assim, o corpo da função consiste de um mapeamento do novo parâmetro para os parâmetros existentes.

class HeatingPlan…

```
xxNEWwithinRange(aNumberRange) {
  return this.withinRange(aNumberRange.low, aNumberRange.high);
```

```
}
```

Agora posso dar início ao trabalho sério, fazendo as chamadas de função atuais chamarem a nova função.

chamada...

```
const low = aRoom.daysTempRange.low;
const high = aRoom.daysTempRange.high;
if (!aPlan.xxNEWwithinRange(aRoom.daysTempRange))
  alerts.push("room temperature went outside range");
```

Ao mudar as chamadas, percebo que parte do código anterior não é mais necessário, portanto lanço mão de *Remover código morto (Remove Dead Code)*.

chamada...

```
~~const low = aRoom.daysTempRange.low;~~
~~const high = aRoom.daysTempRange.high;~~
if (!aPlan.xxNEWwithinRange(aRoom.daysTempRange))
  alerts.push("room temperature went outside range");
```

Faço as substituições, uma de cada vez, testando após cada mudança.

Depois de ter feito todas as substituições, posso usar *Internalizar função (Inline Function) na função* original.

class HeatingPlan…

```
xxNEWwithinRange(aNumberRange) {
  return (aNumberRange.low >= this._temperatureRange.low) &&
    (aNumberRange.high <= this._temperatureRange.high);
}
```

Por fim, removo aquele prefixo feio da nova função de todas as suas chamadas. O prefixo faz com que a operação seja uma substituição global simples, mesmo que eu não tenha um suporte robusto para renomear em meu editor.

class HeatingPlan…

```
withinRange(aNumberRange) {
  return (aNumberRange.low >= this._temperatureRange.low) &&
    (aNumberRange.high <= this._temperatureRange.high);
}
```

chamada...

```
if (!aPlan.withinRange(aRoom.daysTempRange))
  alerts.push("room temperature went outside range");
```

Exemplo: uma variação para criar a nova função

No exemplo anterior, escrevi o código da nova função diretamente. Na maioria das vezes, é muito simples, e é o modo mais fácil de trabalhar. Contudo, há uma variação, que será ocasionalmente conveniente – e que poderá permitir que eu componha a nova função totalmente a partir de refatorações.

Começo com uma chamada da função existente.

chamada...

```
const low = aRoom.daysTempRange.low;
const high = aRoom.daysTempRange.high;
if (!aPlan.withinRange(low, high))
  alerts.push("room temperature went outside range");
```

Quero reorganizar o código de modo a criar a nova função usando *Extrair função (Extract Function)* em um código existente. O código que faz a chamada ainda não está pronto, mas posso prepará-lo usando *Extrair variável (Extract Variable)* algumas vezes. Em primeiro lugar, separo a chamada da função antiga da condicional.

chamada...

```
const low = aRoom.daysTempRange.low;
const high = aRoom.daysTempRange.high;
const isWithinRange = aPlan.withinRange(low, high);
if (!isWithinRange)
  alerts.push("room temperature went outside range");
```

Em seguida, extraio o parâmetro de entrada.

chamada...

```
const tempRange = aRoom.daysTempRange;
const low = tempRange.low;
const high = tempRange.high;
const isWithinRange = aPlan.withinRange(low, high);
if (!isWithinRange)
  alerts.push("room temperature went outside range");
```

Depois de ter feito isso, posso agora usar *Extrair função (Extract Function)* para criar a nova função.

chamada...

```
const tempRange = aRoom.daysTempRange;
const isWithinRange = xxNEWwithinRange(aPlan, tempRange);
```

```
if (!isWithinRange)
  alerts.push("room temperature went outside range");
```

nível mais alto…

```
function xxNEWwithinRange(aPlan, tempRange) {
  const low = tempRange.low;
  const high = tempRange.high;
  const isWithinRange = aPlan.withinRange(low, high);
  return isWithinRange;
}
```

Como a função original está em um contexto diferente (na classe HeatingPlan), tenho de usar *Mover função (Move Function)*.

chamada...

```
const tempRange = aRoom.daysTempRange;
const isWithinRange = aPlan.xxNEWwithinRange(tempRange);
if (!isWithinRange)
  alerts.push("room temperature went outside range");
```

class HeatingPlan…

```
xxNEWwithinRange(tempRange) {
  const low = tempRange.low;
  const high = tempRange.high;
  const isWithinRange = this.withinRange(low, high);
  return isWithinRange;
}
```

Em seguida, continuo como antes, substituindo outras chamadas e internalizando a função antiga na nova. Eu também internalizaria as variáveis que extraí para proporcionar uma separação clara para a extração da nova função.

Como essa variação é totalmente composta de refatorações, ela será particularmente conveniente se eu tiver uma ferramenta de refatoração com operações robustas de extração e de internalização.

Substituir parâmetro por consulta (Replace Parameter with Query)

anteriormente: *Substituir parâmetro por método* (Replace Parameter with Method)

inversa de: *Substituir consulta por parâmetro (Replace Query with*

Parameter)


```
availableVacation(anEmployee,
anEmployee.grade);

function availableVacation(anEmployee, grade)
{
  // calcula férias...
```

```
availableVacation(anEmployee)

function
availableVacation(anEmployee) {
  const grade = anEmployee.grade;
  // calcula férias...
```

Motivação

A lista de parâmetros de uma função deve sintetizar os pontos de variabilidade dessa função, indicando as principais maneiras como ela pode se comportar de modo diferente. Como ocorre com qualquer instrução no código, é bom evitar alguma duplicação, e o código será mais fácil de entender se a lista de parâmetros for pequena.

Se uma chamada passar um valor que a função possa facilmente determinar por conta própria, essa é uma forma de duplicação – uma duplicação que complica desnecessariamente quem faz a chamada, pois esse deve determinar o valor de um parâmetro, quando poderia estar livre dessa tarefa.

O limite para isso é sugerido pela palavra “facilmente”. Ao remover o parâmetro, estou transferindo a responsabilidade por determinar o valor do parâmetro. Se o parâmetro estiver presente, determinar seu valor será responsabilidade de quem faz a chamada; caso contrário, essa responsabilidade será transferida para o corpo da função. Costumo simplificar a vida de quem faz as chamadas, e isso implica transferir a responsabilidade para o corpo da função – mas apenas se essa responsabilidade for apropriada à função.

O motivo mais comum para evitar *Substituir parâmetro por consulta* é se a remoção do parâmetro acrescentar uma dependência indesejada no corpo da função – forçando-o a acessar um elemento de programa que eu preferiria que lhe fosse desconhecido. Talvez fosse uma nova dependência, ou uma dependência já existente que eu gostaria de eliminar. Em geral, isso surge nos pontos em que seria necessário adicionar uma chamada de função problemática no corpo da função, ou acessar algo em um objeto receptor que eu preferiria mover para outro lugar mais tarde.

O caso mais seguro para *Substituir parâmetro por consulta* é quando o valor do parâmetro que quero remover é determinado simplesmente por meio de uma consulta a outro parâmetro da lista. Raramente faria sentido passar dois parâmetros se um deles puder ser determinado a partir do outro.

Um detalhe que exige atenção é se a função que eu estiver considerando tiver transparência referencial – isto é, se eu puder ter certeza de que ela se comportará do mesmo modo sempre que for chamada com os mesmos valores de parâmetros. É muito mais fácil raciocinar com funções como essas e testá-las, e eu não gostaria de alterá-las de modo que perdessem essa propriedade. Assim, eu não substituiria um parâmetro por um acesso a uma variável global mutável.

Procedimento

- Se for necessário, use *Extrair função (Extract Function)* no cálculo do parâmetro.
- Substitua as referências ao parâmetro no corpo da função por referências à expressão que gerem o parâmetro. Teste após cada mudança.
- Use *Mudar declaração de função (Change Function Declaration)* para remover o parâmetro.

Exemplo

Com muita frequência, uso *Substituir parâmetro por consulta* depois de ter realizado outras refatorações que façam com que um parâmetro não seja mais necessário. Considere o código a seguir:

class Order…

```
get finalPrice() {
  const basePrice = this.quantity * this.itemPrice;
```

```
    let discountLevel;
    if (this.quantity > 100) discountLevel = 2;
    else discountLevel = 1;
    return this.discountedPrice(basePrice, discountLevel);
  }
  discountedPrice(basePrice, discountLevel) {
    switch (discountLevel) {
      case 1: return basePrice * 0.95;
      case 2: return basePrice * 0.9;
    }
  }
```

Quando simplifico uma função, tendo a aplicar *Substituir variável temporária por consulta (Replace Temp with Query)*, que resultaria em:

class Order…

```
  get finalPrice() {
    const basePrice = this.quantity * this.itemPrice;
    return this.discountedPrice(basePrice, this.discountLevel);
  }
  get discountLevel() {
    return (this.quantity > 100) ? 2 : 1;
  }
```

Depois de ter feito isso, não haverá mais necessidade de passar o resultado de `discountLevel` para `discountedPrice` – ele mesmo poderá fazer facilmente a chamada.

Substituo qualquer referência ao parâmetro por uma chamada ao método.

class Order…

```
  discountedPrice(basePrice, discountLevel) {
    switch (this.discountLevel) {
      case 1: return basePrice * 0.95;
      case 2: return basePrice * 0.9;
    }
  }
```

Então posso usar *Mudar declaração de função (Change Function Declaration)* para remover o parâmetro.

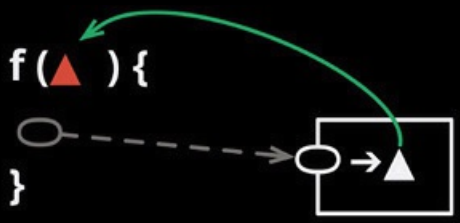
class Order…

```
  get finalPrice() {
    const basePrice = this.quantity * this.itemPrice;
    return this.discountedPrice(basePrice~~, this.discountLevel~~);
  }
  discountedPrice(basePrice~~, discountLevel~~) {
```

```
    switch (this.discountLevel) {
      case 1: return basePrice * 0.95;
      case 2: return basePrice * 0.9;
    }
  }
```

Substituir consulta por parâmetro (Replace Query with Parameter)

inversa de: *Substituir parâmetro por consulta (Replace Parameter with Query)*




```
targetTemperature(aPlan)

function targetTemperature(aPlan) {
  currentTemperature =
thermostat.currentTemperature;
  // restante da função...
```

```
targetTemperature(aPlan,
thermostat.currentTemperature)

function targetTemperature(aPlan,
currentTemperature) {
  // restante da função...
```

Motivação

Ao observar o corpo de uma função, às vezes vejo referências a algo no escopo dela com as quais não me sinto satisfeito. Pode ser uma referência a uma variável global ou a um elemento do mesmo módulo que pretendo mover para outro lugar. Para resolver esse problema, é necessário substituir a referência interna por um parâmetro, transferindo a responsabilidade de resolver a referência para quem chama a função.

A maioria desses casos se deve ao meu desejo de alterar os relacionamentos de dependência no código – para que a função final não seja mais dependente do elemento que quero parametrizar. Há uma tensão, nesse caso, entre converter tudo para parâmetros, o que resultaria em listas de parâmetros longas e repetitivas, e compartilhar bastante escopo, resultando em alto nível de acoplamento entre funções. Como em muitas decisões complicadas, não é algo sobre o qual eu tenha absoluta certeza, portanto é importante que seja possível modificar o código de forma confiável para que o programa tire proveito de minha compreensão cada vez maior.

É mais fácil pensar em uma função que sempre dará o mesmo resultado quando for chamada com os mesmos valores de parâmetros – isso se chama transparência referencial. Se uma função acessar um elemento que não seja referencialmente transparente em seu escopo, a função que a contém também não terá transparência referencial. Posso corrigir isso fazendo com que esse elemento seja um parâmetro. Embora uma ação como essa transfira a responsabilidade para quem faz a chamada, em geral há muitas vantagens em criar módulos claros, com transparência referencial. Um padrão comum é ter módulos que consistem de funções puras, encapsuladas por uma lógica que cuide da E/S e de outros elementos variáveis de um programa. Posso usar *Substituir consulta por parâmetro* para deixar partes de um programa mais puras, facilitando os testes e a compreensão dessas partes.

Contudo, *Substituir consulta por parâmetro* não representa somente vantagens. Ao mover uma consulta para um parâmetro, estarei forçando quem faz a chamada a descobrir como fornecer esse valor. Isso complica a vida de quem chama as funções, e minha tendência habitual é projetar interfaces que facilitem a vida de seus clientes. No final, tudo se reduz à alocação de responsabilidades pelo programa, e essa decisão não é fácil nem imutável – por isso, é necessário conhecer bem essa refatoração (e a sua inversa).

Procedimento

- Use *Extrair variável (Extract Variable)* no código da consulta para separá-lo do restante do corpo da função.
- Aplique *Extrair função (Extract Function)* no código do corpo que não seja a chamada da consulta.
- Dê à nova função um nome fácil de pesquisar, para poder renomeá-la

depois.

- Use *Internalizar variável (Inline Variable)* para se livrar da variável que você acabou de criar.
- Aplique *Internalizar função (Inline Function)* na função original.
- Renomeie a nova função dando-lhe o nome da função original.

Exemplo

Considere um sistema de controle de temperatura simples, porém irritante. Ele permite ao usuário selecionar uma temperatura em um termostato – mas define a temperatura final somente em um intervalo determinado por um plano de aquecimento (heating plan).

class HeatingPlan…

```
get targetTemperature() {
  if (thermostat.selectedTemperature > this._max) return this._max;
  else if (thermostat.selectedTemperature < this._min) return this._min;
  else return thermostat.selectedTemperature;
}
```

chamada...

```
if (thePlan.targetTemperature > thermostat.currentTemperature) setToHeat();
else if (thePlan.targetTemperature < thermostat.currentTemperature) setToCool();
else setOff();
```

Como usuário de um sistema desse tipo, posso ficar irritado com o fato de meus desejos serem suplantados pelas regras do plano de aquecimento, mas, como programador, talvez esteja mais preocupado com o fato de a função `targetTemperature` apresentar uma dependência em relação a um objeto de termostato global. Posso acabar com essa dependência fazendo com que ele seja um parâmetro.

Meu primeiro passo é usar *Extrair variável (Extract Variable)* no parâmetro que eu gostaria de ter em minha função.

class HeatingPlan…

```
get targetTemperature() {
  const selectedTemperature = thermostat.selectedTemperature;
  if (selectedTemperature > this._max) return this._max;
  else if (selectedTemperature < this._min) return this._min;
  else return selectedTemperature;
}
```

Isso facilita aplicar *Extrair função (Extract Function)* no corpo todo da função, exceto na parte que determina o parâmetro.

class HeatingPlan…

```
get targetTemperature() {
  const selectedTemperature = thermostat.selectedTemperature;
  return this.xxNEWtargetTemperature(selectedTemperature);
}
xxNEWtargetTemperature(selectedTemperature) {
  if (selectedTemperature > this._max) return this._max;
  else if (selectedTemperature < this._min) return this._min;
  else return selectedTemperature;
}
```

Em seguida, internalizo a variável que acabei de extrair, fazendo com que a função seja uma chamada simples.

class HeatingPlan…

```
get targetTemperature() {
  return this.xxNEWtargetTemperature(thermostat.selectedTemperature);
}
```

Agora posso usar *Internalizar função (Inline Function)* nesse método.

chamada...

```
if (thePlan.xxNEWtargetTemperature(thermostat.selectedTemperature) >
    thermostat.currentTemperature)
  setToHeat();
else if (thePlan.xxNEWtargetTemperature(thermostat.selectedTemperature) <
    thermostat.currentTemperature)
  setToCool();
else
  setOff();
```

Tiro proveito do nome da nova função, o qual pode ser facilmente pesquisado, e renomeio removendo o prefixo.

chamada...

```
if (thePlan.targetTemperature(thermostat.selectedTemperature) >
    thermostat.currentTemperature)
  setToHeat();
else if (thePlan.targetTemperature(thermostat.selectedTemperature) <
    thermostat.currentTemperature)
  setToCool();
else
```

```
    setOff();
```

class HeatingPlan…

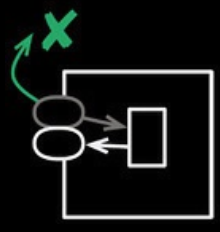
```
  targetTemperature(selectedTemperature) {
    if (selectedTemperature > this._max) return this._max;
    else if (selectedTemperature < this._min) return this._min;
    else return selectedTemperature;
  }
```

Como ocorre com frequência nessa refatoração, o código que faz a chamada parece mais complicado que antes. Mover uma dependência para fora de um módulo transfere a responsabilidade de lidar com essa dependência para quem faz a chamada. É o preço a ser pago pela redução de acoplamento.

No entanto, eliminar o acoplamento com o objeto termostato não é o único benefício resultante dessa refatoração. A classe `HeatingPlan` é imutável – seus campos são definidos no construtor e não há métodos para alterá-los. (Pouparei você do esforço de analisar a classe inteira; apenas acredite em mim.) Considerando um plano de aquecimento imutável, ao mover a referência ao termostato para fora do corpo da função, também deixei `targetTemperature` referencialmente transparente. Sempre que eu chamar `targetTemperature` no mesmo objeto, com o mesmo argumento, obterei o mesmo resultado. Se todos os métodos do plano de aquecimento tiverem transparência referencial, essa classe será muito mais fácil de testar, além de ser muito mais fácil entendê-la.

Um problema com o modelo de classes de JavaScript é que é impossível garantir que uma classe seja imutável – há sempre um meio de acessar os dados de um objeto. Contudo, escrever uma classe que sinalize e incentive a imutabilidade em geral é suficiente. Criar classes que tenham essa característica com frequência é uma estratégia robusta, e *Substituir consulta por parâmetro* é uma ferramenta conveniente para isso.

Remover método de escrita (Remove Setting Method)



```
class Person {
  get name() {...}
  set name(aString)
{...}
```

```
class Person {
  get name()
{...}
```

Motivação

Disponibilizar um método de escrita indica que um campo pode ser alterado. Se eu não quiser que esse campo mude depois que o objeto for criado, não disponibilizarei um método de escrita (e tornarei o campo imutável). Desse modo, o campo recebe valor somente no construtor, minha intenção de não modificar esse campo estará clara e, geralmente, eliminarei a própria possibilidade de que o campo mude.

Há alguns casos comuns em que essa situação pode surgir. Um deles é aquele em que as pessoas sempre usam um método de acesso para manipular um campo, mesmo nos construtores. Isso resulta em uma única chamada a um método de escrita sendo feita no construtor. Prefiro remover o método de escrita para deixar claro que as atualizações não fazem sentido após a construção.

Outro caso é aquele em que o objeto é criado por clientes usando um script de criação no lugar de uma simples chamada de construtor. Um script de criação como esse começa com a chamada do construtor, seguida de uma sequência de chamadas de métodos de escrita para criar o novo objeto. Depois que o script termina, não se espera que o novo objeto altere certos campos (ou todos). É esperado que os setters sejam chamados somente durante essa criação inicial. Nesse caso, eu me livraria deles para deixar minhas intenções mais claras.

Procedimento

- Se o valor sendo configurado não for fornecido ao construtor, use *Mudar declaração de função (Change Function Declaration)* para adicioná-lo. Acrescente uma chamada ao método de escrita no construtor.

Se quiser remover vários métodos de escrita, acrescente todos os valores no construtor de uma só vez. Isso simplificará os passos futuros.

- Remova cada chamada a um método de escrita fora do construtor usando o novo valor do construtor. Teste após cada mudança.

Se não for possível substituir a chamada ao setter criando um novo objeto (porque você está atualizando um objeto de referência compartilhado), abandone a refatoração.

- Use *Internalizar função (Inline Function)* no método de escrita. Se possível, deixe o campo imutável.
- Teste.

Exemplo

Tenho uma classe simples de pessoa (person).

class Person…

```
get name() {return this._name;}
set name(arg) {this._name = arg;}
get id() {return this._id;}
set id(arg) {this._id = arg;}
```

Atualmente, crio um novo objeto com um código como este:

```
const martin = new Person();
martin.name = "martin";
martin.id = "1234";
```

O nome de uma pessoa pode mudar depois que ela é criada, mas não o ID. Para deixar isso claro, gostaria de remover o método de escrita no ID.

Ainda é necessário definir o ID no início, portanto usarei *Mudar declaração de função (Change Function Declaration)* para adicioná-lo no construtor.

class Person…

```
constructor(id) {
  this.id = id;
}
```

Em seguida, faço adaptações no script de criação para que o ID seja definido no construtor.

```
const martin = new Person("1234");
martin.name = "martin";
~~martin.id = "1234";~~
```

Faço isso em todos os lugares em que uma pessoa é criada, testando após cada mudança.

Quando terminar, posso aplicar *Internalizar função (Inline Function)* no método de escrita.

class Person…

```
constructor(id) {
  this._id = id;
}
get name() {return this._name;}
set name(arg) {this._name = arg;}
get id() {return this._id;}
set id(arg) {this._id = arg;}
```

Substituir construtor por função de factory (Replace Constructor with Factory Function)

anteriormente: *Substituir construtor por método de factory* (Replace Constructor with Factory Method)


```
leadEngineer = new Employee(document.leadEngineer,
'E');
```

```
leadEngineer =
createEngineer(document.leadEngineer);
```

Motivação

Muitas linguagens orientadas a objetos têm uma função especial de construtor, chamada para inicializar um objeto. Os clientes geralmente chamam esse construtor quando querem criar um novo objeto. Entretanto, esses construtores muitas vezes apresentam limitações inconvenientes, que não estão presentes em funções mais genéricas. Um construtor Java deve devolver uma instância da classe com a qual foi chamado, o que significa que

não poderei substituí-lo por uma subclasse ou um proxy, dependendo do ambiente ou dos parâmetros. A nomenclatura do construtor é fixa, tornando impossível para mim usar um nome que seja mais claro que o nome default. Com frequência, os construtores exigem um operador especial para serem chamados (“new” em muitas linguagens), o que dificulta usá-los em contextos que esperem funções comuns.

Uma função de factory não está sujeita a essas limitações. Provavelmente ela chamará o construtor como parte de sua implementação, mas posso substituí-lo livremente por outro código.

Procedimento

- Crie uma função de factory, na qual o corpo seja uma chamada para o construtor.
- Substitua cada chamada ao construtor por uma chamada à função de factory.
- Teste após cada mudança.
- Limite a visibilidade do construtor o máximo possível.

Exemplo

Um exemplo rápido, embora tedioso, utiliza tipos de funcionários (employees). Considere uma classe funcionário como esta:

class Employee…

```
constructor (name, typeCode) {
  this._name = name;
  this._typeCode = typeCode;
}
get name() {return this._name;}
get type() {
  return Employee.legalTypeCodes[this._typeCode];
}
static get legalTypeCodes() {
  return {"E": "Engineer", "M": "Manager", "S": "Salesman"};
}
```

Ela é usada em:

chamada...

```
candidate = new Employee(document.name, document.empType);
```

e em:

chamada...

```
const leadEngineer = new Employee(document.leadEngineer, 'E');
```

Meu primeiro passo é criar a função de factory. Seu corpo contém uma simples delegação para o construtor.

nível mais alto…

```
function createEmployee(name, typeCode) {
  return new Employee(name, typeCode);
}
```

Em seguida, localizo as chamadas ao construtor e as altero, uma de cada vez, para que usem a função de factory em seu lugar.

A primeira chamada é óbvia:

chamada...

```
candidate = createEmployee(document.name, document.empType);
```

No segundo caso, eu poderia usar a nova função de factory assim:

chamada...

```
const leadEngineer = createEmployee(document.leadEngineer, 'E');
```

Entretanto, não gosto de usar o código de tipo nesse caso – em geral, passar um código como uma string literal constitui um mau cheiro no código. Portanto, prefiro criar uma nova função de factory que inclua em seu nome o tipo do funcionário desejado.

chamada...

```
const leadEngineer = createEngineer(document.leadEngineer);
```

nível mais alto…

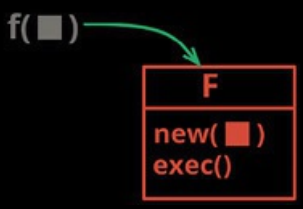
```
function createEngineer(name) {
  return new Employee(name, 'E');
}
```

Substituir função por comando (Replace Function with Command)

anteriormente: *Substituir método por objeto de método* (Replace Method with Method Object)

inversa de: *Substituir comando por função (Replace Command with*

Function)




```
function score(candidate, medicalExam,
scoringGuide) {
  let result = 0;
  let healthLevel = 0;
  // código longo do corpo
}
```

```
class Scorer {
  constructor(candidate, medicalExam,
scoringGuide) {
    this._candidate = candidate;
    this._medicalExam = medicalExam;
    this._scoringGuide = scoringGuide;
  }

  execute() {
    this._result = 0;
    this._healthLevel = 0;
    // código longo do corpo
  }
}
```

Motivação

As funções – sejam elas independentes ou associadas a objetos na forma de métodos – são um dos blocos de construção fundamentais da programação. Há ocasiões, porém, em que é conveniente encapsular uma função em seu próprio objeto, ao qual me refiro como um “objeto de comando”, ou simplesmente um comando. Um objeto como esse, na maioria dos casos, é construído em torno de um único método, cuja requisição e execução são o propósito do objeto.

Um comando oferece mais flexibilidade para o controle e a expressão de

uma função, em comparação com o mecanismo simples de função. Os comandos podem ter operações complementares, por exemplo, uma operação de desfazer (undo). Posso disponibilizar métodos para construir seus parâmetros a fim de oferecer um suporte para um ciclo de vida mais rico, além de incluir personalizações usando herança e ganchos (hooks). Se eu estiver trabalhando em uma linguagem com objetos, mas sem funções de primeira classe, posso oferecer boa parte dessas funcionalidades usando comandos em seu lugar. De modo semelhante, posso usar métodos e campos para ajudar a dividir uma função complexa, mesmo em uma linguagem que não tenha funções aninhadas, e posso chamar esses métodos diretamente enquanto faço testes e depuração.

Tudo isso são bons motivos para usar comandos, e tenho de estar preparado para refatorar funções e transformá-las em comandos quando necessário. Entretanto, não devemos esquecer que essa flexibilidade, como sempre, tem um preço a ser pago em termos de complexidade. Assim, dada a opção de escolher entre uma função de primeira classe e um comando, optarei pela função em 95% das vezes. Uso um comando apenas se precisar especificamente de um recurso que abordagens mais simples não são capazes de oferecer.

Como muitas palavras em desenvolvimento de software, “comando” tem vários significados. No contexto em que estou usando aqui, o comando é um objeto que encapsula uma requisição, seguindo o padrão de comando dos Padrões de Projeto (Design Patterns) [gof]. Quando uso “comando” nesse sentido, utilizo “objeto de comando” para definir o contexto, e “comando” depois. A palavra “comando” também é usada no princípio da separação entre comando-consulta [mf-cqs], em que um comando é um método de objeto que modifica um estado observável. Sempre procuro evitar usar comando nesse sentido, preferindo “modificador” ou “mutator”.

Procedimento

- Crie uma classe vazia para a função. Dê-lhe um nome baseado na função.
- Use *Mover função (Move Function)* a fim de mover a função para a classe vazia.

Mantenha a função original como uma função de encaminhamento, pelo menos até o término da refatoração.

Siga qualquer convenção que a linguagem tenha para nomear comandos. Se

não houver nenhuma convenção, escolha um nome genérico para a função de execução do comando, por exemplo, “execute” ou “call”.

- Considere criar um campo para cada argumento, e mova esses argumentos para o construtor.

Exemplo

A linguagem JavaScript tem muitas deficiências, mas uma de suas grandes decisões foi fazer com que as funções sejam entidades de primeira classe. Desse modo, não preciso passar por todos os obstáculos envolvendo a criação de comandos para tarefas comuns, que existem nas linguagens sem esse recurso. Contudo, ainda há ocasiões em que um comando é a ferramenta correta para a tarefa.

Um desses casos é na separação de uma função complexa, de modo a compreendê-la e modificá-la mais facilmente. Para mostrar de fato a importância dessa refatoração, preciso de uma função longa e complicada – mas ela seria muito extensa para escrever, e mais difícil ainda para você ler. Em vez disso, optei por uma função pequena, que não precisaria dessa refatoração. Essa função atribui pontos (scores) para uma aplicação de seguro (insurance application):

```
function score(candidate, medicalExam, scoringGuide) {
  let result = 0;
  let healthLevel = 0;
  let highMedicalRiskFlag = false;

  if (medicalExam.isSmoker) {
    healthLevel += 10;
    highMedicalRiskFlag = true;
  }
  let certificationGrade = "regular";
  if (scoringGuide.stateWithLowCertification(candidate.originState)) {
    certificationGrade = "low";
    result -= 5;
  }
  // muito mais código parecido com esse
  result -= Math.max(healthLevel - 5, 0);
  return result;
}
```

Começo criando uma classe vazia e, então, uso *Mover função (Move Function)* a fim de mover a função para essa classe.

```
function score(candidate, medicalExam, scoringGuide) {
  return new Scorer().execute(candidate, medicalExam, scoringGuide);
}
class Scorer {
  execute (candidate, medicalExam, scoringGuide) {
    let result = 0;
    let healthLevel = 0;
    let highMedicalRiskFlag = false;

    if (medicalExam.isSmoker) {
      healthLevel += 10;
      highMedicalRiskFlag = true;
    }
    let certificationGrade = "regular";
    if (scoringGuide.stateWithLowCertification(candidate.originState)) {
      certificationGrade = "low";
      result -= 5;
    }
    // muito mais código parecido com esse
    result -= Math.max(healthLevel - 5, 0);
    return result;
  }
}
```

Na maioria das vezes, prefiro passar argumentos para um comando no construtor e ter o método execute sem parâmetros. Embora isso seja menos importante para um cenário de decomposição simples como este, será muito conveniente se eu quiser manipular o comando que tenha um ciclo de vida mais complicado para definição de parâmetros ou com personalizações. Diferentes classes de comando podem ter parâmetros distintos, mas os comandos poderão estar misturados quando forem enfileirados para execução.

Posso cuidar desses parâmetros, um de cada vez.

```
function score(candidate, medicalExam, scoringGuide) {
  return new Scorer(candidate).execute(~~candidate,~~ medicalExam, scoringGuide);
}
```

class Scorer…

```
constructor(candidate){
  this._candidate = candidate;
}

execute (~~candidate,~~ medicalExam, scoringGuide) {
  let result = 0;
```

```
    let healthLevel = 0;
    let highMedicalRiskFlag = false;

    if (medicalExam.isSmoker) {
      healthLevel += 10;
      highMedicalRiskFlag = true;
    }
    let certificationGrade = "regular";
    if (scoringGuide.stateWithLowCertification(this._candidate.originState)) {
      certificationGrade = "low";
      result -= 5;
    }
    // muito mais código parecido com esse
    result -= Math.max(healthLevel - 5, 0);
    return result;
  }
```

Prossigo com os demais parâmetros:

```
  function score(candidate, medicalExam, scoringGuide) {
    return new Scorer(candidate, medicalExam, scoringGuide).execute();
  }
```

class Scorer…

```
  constructor(candidate, medicalExam, scoringGuide){
    this._candidate = candidate;
    this._medicalExam = medicalExam;
    this._scoringGuide = scoringGuide;
  }

  execute () {
    let result = 0;
    let healthLevel = 0;
    let highMedicalRiskFlag = false;

    if (this._medicalExam.isSmoker) {
      healthLevel += 10;
      highMedicalRiskFlag = true;
    }
    let certificationGrade = "regular";
    if (this._scoringGuide.stateWithLowCertification(this._candidate.originState)) {
      certificationGrade = "low";
      result -= 5;
    }
    // muito mais código parecido com esse
    result -= Math.max(healthLevel - 5, 0);
    return result;
```

```
}
```

Com isso, concluo *Substituir função por comando*, mas o ponto principal ao fazer essa refatoração é permitir que eu divida funções complicadas – portanto, deixe-me descrever alguns passos para fazer isso. Meu próximo passo, nesse caso, é passar todas as variáveis locais para campos. Novamente, faço isso, uma variável de cada vez.

class Scorer…

```
constructor(candidate, medicalExam, scoringGuide){
  this._candidate = candidate;
  this._medicalExam = medicalExam;
  this._scoringGuide = scoringGuide;
}

execute () {
  this._result = 0;
  let healthLevel = 0;
  let highMedicalRiskFlag = false;

  if (this._medicalExam.isSmoker) {
    healthLevel += 10;
    highMedicalRiskFlag = true;
  }
  let certificationGrade = "regular";
  if (this._scoringGuide.stateWithLowCertification(this._candidate.originState)) {
    certificationGrade = "low";
    this._result -= 5;
  }
  // muito mais código parecido com esse
  this._result -= Math.max(healthLevel - 5, 0);
  return this._result;
}
```

Repito esse procedimento para todas as variáveis locais. (Essa é uma daquelas refatorações que achei ser suficientemente simples, a ponto de não ter criado uma entrada para ela no catálogo. Sinto-me um pouco culpado por isso.)

class Scorer…

```
constructor(candidate, medicalExam, scoringGuide){
  this._candidate = candidate;
  this._medicalExam = medicalExam;
  this._scoringGuide = scoringGuide;
}
```

```
execute () {
  this._result = 0;
  this._healthLevel = 0;
  this._highMedicalRiskFlag = false;

  if (this._medicalExam.isSmoker) {
    this._healthLevel += 10;
    this._highMedicalRiskFlag = true;
  }
  this._certificationGrade = "regular";
  if (this._scoringGuide.stateWithLowCertification(this._candidate.originState)) {
    this._certificationGrade = "low";
    this._result -= 5;
  }
  // muito mais código parecido com esse
  this._result -= Math.max(this._healthLevel - 5, 0);
  return this._result;
}
```

Agora que movi todo o estado da função para o objeto de comando, posso usar refatorações como *Extrair função (Extract Function)*, sem me deixar envolver na confusão criada por todas as variáveis e seus escopos.

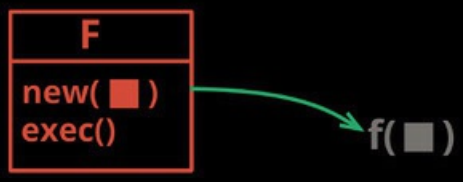
class Scorer…

```
execute () {
  this._result = 0;
  this._healthLevel = 0;
  this._highMedicalRiskFlag = false;

  this.scoreSmoking();
  this._certificationGrade = "regular";
  if (this._scoringGuide.stateWithLowCertification(this._candidate.originState)) {
    this._certificationGrade = "low";
    this._result -= 5;
  }
  // muito mais código parecido com esse
  this._result -= Math.max(this._healthLevel - 5, 0);
  return this._result;
  }
scoreSmoking() {
  if (this._medicalExam.isSmoker) {
    this._healthLevel += 10;
    this._highMedicalRiskFlag = true;
  }
}
```

Isso me permite tratar o comando de modo semelhante à forma como eu lidaria com uma função aninhada. Com efeito, ao fazer essa refatoração em JavaScript, usar funções aninhadas seria uma alternativa razoável a usar um comando. Eu ainda preferiria usar um comando nesse caso, em parte porque tenho mais familiaridade com eles, e em parte porque, com um comando, posso escrever testes e depurar chamadas com base nas subfunções.

Substituir comando por função (Replace Command with Function)

inversa de: *Substituir função por comando (Replace Function with Command)*




```
class ChargeCalculator {
  constructor (customer, usage){
    this._customer = customer;
    this._usage = usage;
  }
  execute() {
    return this._customer.rate *
this._usage;
  }
}
```

```
function charge(customer,
usage) {
  return customer.rate * usage;
}
```

Motivação

Objetos de comando oferecem um mecanismo eficaz para lidar com processamentos complexos. Esses podem ser facilmente divididos em

métodos separados que compartilhem um estado comum por meio de campos; podem ser chamados por métodos diferentes para ter efeitos diferentes; podem ter seus dados construídos em etapas. Contudo, essa eficácia tem um custo. Na maioria das vezes, quero apenas chamar uma função para que ela faça o que tenha de ser feito. Se for esse o caso, e a função não for muito complexa, não valerá a pena ter um objeto de comando, pois ele representará mais problemas, e a função deverá ser transformada em uma função comum.

Procedimento

- Aplique *Extrair função (Extract Function)* na criação do comando e na chamada ao método de execução do comando.

Com isso, a nova função será criada, a qual substituirá o comando no devido tempo.

- Para cada método chamado pelo método de execução do comando, aplique *Internalizar função (Inline Function)*.

Se a função auxiliar devolver um valor, use *Extrair variável (Extract Variable)* na chamada antes, e depois utilize *Internalizar função (Inline Function)*.

- Use *Mudar declaração de função (Change Function Declaration)* para colocar todos os parâmetros do construtor no método de execução do comando.
- Para cada campo, altere as referências no método de execução do comando para que usem o parâmetro. Teste após cada mudança.
- Internalize a chamada do construtor e a chamada ao método de execução do comando no ponto de chamada (que é a função que fará a substituição).
- Teste.
- Aplique *Remover código morto (Remove Dead Code)* na classe de comando.

Exemplo

Começarei com o pequeno objeto de comando a seguir:

```
class ChargeCalculator {
  constructor (customer, usage, provider){
```

```
    this._customer = customer;
    this._usage = usage;
    this._provider = provider;
  }
  get baseCharge() {
    return this._customer.baseRate * this._usage;
  }
  get charge() {
    return this.baseCharge + this._provider.connectionCharge;
  }
}
```

Ele é usado por um código como este:

chamada...

```
monthCharge = new ChargeCalculator(customer, usage, provider).charge;
```

A classe de comando é pequena e simples o bastante para ser mais apropriada como uma função.

Começo usando *Extrair função (Extract Function)* para encapsular a criação da classe e a chamada.

chamada...

```
monthCharge = charge(customer, usage, provider);
```

nível mais alto…

```
function charge(customer, usage, provider) {
  return new ChargeCalculator(customer, usage, provider).charge;
}
```

Tenho de decidir como lidar com qualquer função auxiliar – nesse caso, com `baseCharge`. Minha abordagem usual para uma função que devolva um valor é usar *Extrair variável (Extract Variable)* antes nesse valor.

class ChargeCalculator…

```
get baseCharge() {
  return this._customer.baseRate * this._usage;
}
get charge() {
  const baseCharge = this.baseCharge;
  return baseCharge + this._provider.connectionCharge;
}
```

Em seguida, uso *Internalizar função (Inline Function)* na função auxiliar.

class ChargeCalculator…

```
  get charge() {
    const baseCharge = this._customer.baseRate * this._usage;
    return baseCharge + this._provider.connectionCharge;
  }
```

Agora tenho todo o processamento em uma única função, portanto meu próximo passo é mover os dados passados para o construtor para o método principal. Inicialmente uso *Mudar declaração de função (Change Function Declaration)* para adicionar todos os parâmetros do construtor no método charge.

class ChargeCalculator…

```
  constructor (customer, usage, provider){
    this._customer = customer;
    this._usage = usage;
    this._provider = provider;
  }
  charge(customer, usage, provider) {
    const baseCharge = this._customer.baseRate * this._usage;
    return baseCharge + this._provider.connectionCharge;
  }
```

nível mais alto…

```
  function charge(customer, usage, provider) {
    return new ChargeCalculator(customer, usage, provider)
                        .charge(customer, usage, provider);
  }
```

Agora posso alterar o corpo de charge para que use os parâmetros passados. Isso pode ser feito, um parâmetro de cada vez.

class ChargeCalculator…

```
  constructor (customer, usage, provider){
    this._customer = customer;
    this._usage = usage;
    this._provider = provider;
  }
  charge(customer, usage, provider) {
    const baseCharge = customer.baseRate * this._usage;
    return baseCharge + this._provider.connectionCharge;
  }
```

Não é necessário remover a atribuição a this._customer no construtor, pois ele será simplesmente ignorado. No entanto, prefiro fazê-lo porque um teste falhará se eu me esquecer de mudar o uso de um campo para o parâmetro. (E,

se um teste não falhar, devo considerar o acréscimo de um novo teste.)

Repito esse procedimento para os outros parâmetros; eis o meu código final:

class ChargeCalculator…

```
charge(customer, usage, provider) {
  const baseCharge = customer.baseRate * usage;
  return baseCharge + provider.connectionCharge;
}
```

Depois de ter feito tudo isso, posso realizar uma internalização na função `charge` de nível mais alto. Esse é um tipo especial de *Internalizar função (Inline Function)*, pois internaliza tanto o construtor quanto a chamada do método.

nível mais alto…

```
function charge(customer, usage, provider) {
    const baseCharge = customer.baseRate * usage;
    return baseCharge + provider.connectionCharge;
}
```

A classe de comando agora é um código morto, portanto usarei *Remover código morto (Remove Dead Code)* para lhe conceder uma cerimônia fúnebre honrada.

CAPÍTULO 12

Lidando com herança

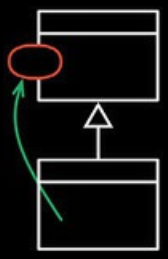
Neste último capítulo, voltarei minha atenção a um dos recursos mais conhecidos da programação orientada a objetos: a herança. Como qualquer mecanismo eficaz, a herança é muito útil, mas também fácil de ser usada indevidamente, e, com frequência, é difícil ver esse uso indevido antes que seja tarde demais.

Muitas vezes, é necessário mover funcionalidades para cima ou para baixo na hierarquia de herança. Várias refatorações lidam com isso: *Subir método (Pull Up Method)*, *Subir campo (Pull Up Field)*, *Subir corpo do construtor (Pull Up Constructor Body)*, *Descer método (Push Down Method)* e *Descer campo (Push Down Field)*. Posso adicionar e remover classes da hierarquia com *Extrair superclasse (Extract Superclass)*, *Remover subclasse (Remove Subclass)* e *Condensar Hierarquia (Collapse Hierarchy)*. Posso acrescentar uma subclasse a fim de substituir um campo que estou usando para acionar um comportamento distinto com base em seu valor; faço isso com *Substituir código de tipos por subclasses (Replace Type Code with Subclasses)*.

A herança é uma ferramenta eficaz: às vezes, porém, ela acaba sendo usada no lugar errado – ou o local em que ela é usada torna-se inapropriado. Nesse caso, uso *Substituir subclasse por delegação (Replace Subclass with Delegate)* ou *Substituir superclasse por delegação (Replace Superclass with Delegate)* para transformar a herança em delegação.

Subir método (Pull Up Method)

inversa de: *Descer método (Push Down Method)*



```
class Employee {...}

class Salesman extends
Employee {
  get name() {...}
}

class Engineer extends
Employee {
  get name() {...}
}
```

```
class Employee {
  get name() {...}
}

class Salesman extends Employee
{...}
class Engineer extends Employee
{...}
```

Motivação

Eliminar código duplicado é importante. Dois métodos duplicados podem funcionar bem como estão, mas nada mais serão além de um terreno fértil para bugs no futuro. Sempre que houver duplicação, haverá risco de uma alteração em uma cópia não ser feita na outra. Em geral, é difícil encontrar as duplicatas.

É mais fácil usar *Subir método* quando os métodos têm o mesmo corpo, o que implica que houve uma operação de copiar e colar. É claro que nem sempre a situação é assim tão evidente. Eu poderia apenas fazer a refatoração e ver se os testes acusam erro – mas isso significa depositar muita confiança em meus testes. Em geral, acho importante procurar as diferenças – muitas

vezes elas exibem comportamentos para os quais esqueci de criar testes.

Frequentemente, *Subir método* é aplicada depois de outros passos. Vejo dois métodos em classes diferentes, que podem ser parametrizados de tal modo que acabam sendo, essencialmente, o mesmo método. Nesse caso, o passo mais rápido será aplicar *Parametrizar função (Parameterize Function)* separadamente e, em seguida, *Subir método*.

A pior das complicações com *Subir método* ocorre se o corpo do método referenciar recursos que estão na subclasse, mas não na superclasse. Se isso acontecer, tenho de usar *Subir campo (Pull Up Field)* e *Subir método* nesses elementos antes.

Se eu tiver dois métodos com um fluxo geral semelhante, mas que diferem quanto a detalhes, considerarei o uso de *Formar método de template* (Form Template Method) [mf-ft].

Procedimento

- Inspecione os métodos para garantir que sejam idênticos.

Se fizerem a mesma tarefa, mas não forem idênticos, refatore-os até que tenham corpos idênticos.

- Verifique se todas as chamadas de métodos e referências a campos no corpo do método fazem referências a recursos que possam ser chamados a partir da superclasse.
- Se os métodos tiverem assinaturas diferentes, use *Mudar declaração de função (Change Function Declaration)* para que fiquem de acordo com o que você quer usar na superclasse.
- Crie um novo método na superclasse. Copie aí o corpo de um dos métodos.
- Execute verificações estáticas.
- Apague o método de uma subclasse.
- Teste.
- Continue apagando os métodos das subclasses até que todos tenham sido removidos.

Exemplo

Tenho dois métodos de subclasse que fazem a mesma tarefa.

class Employee extends Party…

```
get annualCost() {
  return this.monthlyCost * 12;
}
```

class Department extends Party…

```
get totalAnnualCost() {
  return this.monthlyCost * 12;
}
```

Observo as duas classes e percebo que elas referenciam a propriedade monthlyCost que não está definida na superclasse, mas está presente nas duas subclasses. Como estou em uma linguagem dinâmica, não há problemas; se estivesse em uma linguagem estática, seria necessário definir um método abstrato em Party.

Os métodos têm nomes diferentes, portanto uso *Mudar declaração de função (Change Function Declaration)* para que tenham o mesmo nome.

class Department…

```
get annualCost() {
  return this.monthlyCost * 12;
}
```

Copio o método de uma subclasse e colo na superclasse.

class Party…

```
get annualCost() {
  return this.monthlyCost * 12;
}
```

Em uma linguagem estática, eu compilaria para garantir que não há problemas com nenhuma referência. Isso não me ajudará no meu caso, portanto inicialmente removo annualCost de Employee, testo e, em seguida, eu o removo de Department.

Com isso, concluo a refatoração, mas resta uma questão. annualCost chama monthlyCost, mas monthlyCost não aparece na classe Party. Tudo funciona porque JavaScript é uma linguagem dinâmica – mas é importante sinalizar que subclasses de Party devem disponibilizar uma implementação para monthlyCost, particularmente se outras subclasses forem acrescentadas depois. Uma boa maneira de fornecer essa indicação é ter um método de armadilha como este:

class Party…

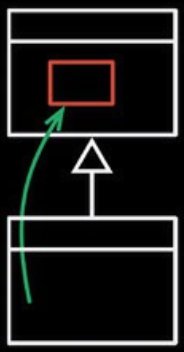
```
get monthlyCost() {
```

```
    throw new SubclassResponsibilityError();
  }
```

Chamo um erro como esse de erro de responsabilidade da subclasse (subclass responsibility error), pois esse era o nome usado em Smalltalk.

Subir campo (Pull Up Field)

inversa de: *Descer campo (Push Down Field)*



```
class Employee {...} // Java

class Salesman extends
Employee {
  private String name;
}

class Engineer extends
Employee {
  private String name;
}
```

```
class Employee {
  protected String name;
}

class Salesman extends Employee
{...}
class Engineer extends Employee
{...}
```

Motivação

Se as subclasses forem desenvolvidas de forma independente, ou se forem combinadas por meio de refatoração, muitas vezes percebo que há duplicação de funcionalidades. Em particular, determinados campos podem estar duplicados. Campos como esses às vezes têm nomes semelhantes – mas nem sempre. A única maneira pela qual posso dizer o que está acontecendo é observar os campos e analisar como estão sendo usados. Se forem usados de modo semelhante, posso subi-los para a superclasse.

Ao fazer isso, reduzo a duplicação de duas formas. Removo a declaração de dados duplicada e posso, então, mover comportamentos que usem o campo, das subclasses para a superclasse.

Muitas linguagens dinâmicas não precisam definir campos como parte da definição de suas classes – em vez disso, os campos passam a existir quando recebem valor pela primeira vez. Nesse caso, subir um campo é essencialmente uma consequência de *Subir corpo do construtor (Pull Up Constructor Body)*.

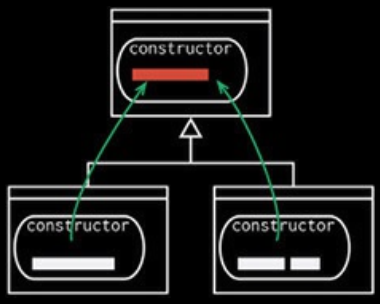
Procedimento

- Inspecione todos os usuários do campo candidato a fim de garantir que estão sendo usados do mesmo modo.
- Se os campos tiverem nomes diferentes, use *Renomear campo (Rename Field)* para lhes dar o mesmo nome.
- Crie um novo campo na superclasse.

O novo campo deverá ser acessível às subclasses (protected em linguagens comuns).

- Apague os campos nas subclasses.
- Teste.

Subir corpo do construtor (Pull Up Constructor Body)




```
class Party {...}

class Employee extends Party {
  constructor(name, id,
monthlyCost) {
    super();
    this._id = id;
    this._name = name;
    this._monthlyCost =
monthlyCost;
  }
}
```

```
class Party {
  constructor(name){
    this._name = name;
  }
}

class Employee extends Party {
  constructor(name, id,
monthlyCost) {
    super(name);
    this._id = id;
    this._monthlyCost =
monthlyCost;
  }
}
```

Motivação

Construtores são elementos ardilosos. Eles não são métodos muito comuns – portanto há mais restrições quanto ao que posso fazer com eles.

Se vejo métodos com um comportamento comum nas subclasses, minha primeira ideia é usar *Extrair função (Extract Function)*, seguida de *Subir método (Pull Up Method)*, movendo o código tranquilamente para a superclasse. Os construtores complicam isso – porque eles têm regras especiais sobre o que pode ser feito e em qual ordem, portanto uma abordagem um pouco diferente é necessária.

Se essa refatoração começar a ficar confusa, lanço mão de *Substituir construtor por função de factory (Replace Constructor with Factory Function)*.

Procedimento

- Defina um construtor na superclasse, caso ainda não exista um. Garanta que ele seja chamado pelos construtores das subclasses.
- Use *Deslocar instruções (Slide Statements)* para mover quaisquer instruções comuns para logo depois da chamada a `super()`.
- Remova o código comum de cada subclasse e coloque-o na superclasse. Adicione quaisquer parâmetros de construtor referenciados no código comum à chamada de `super()`.
- Teste.
- Se houver algum código comum que não possa ser movido para o início do construtor, use *Extrair função (Extract Function)*, seguido de *Subir método (Pull Up Method)*.

Exemplo

Começarei com o código a seguir:

```
class Party {}
class Employee extends Party {
  constructor(name, id, monthlyCost) {
    super();
    this._id = id;
    this._name = name;
    this._monthlyCost = monthlyCost;
  }
  // restante da classe...
class Department extends Party {
  constructor(name, staff){
```

```
    super();
    this._name = name;
    this._staff = staff;
  }
  // restante da classe...
```

O código comum, nesse exemplo, é a atribuição do nome. Uso *Deslocar instruções (Slide Statements)* para mover a atribuição em `Employee` para próximo da chamada a `super()`:

```
class Employee extends Party {
  constructor(name, id, monthlyCost) {
    super();
    this._name = name;
    this._id = id;
    this._monthlyCost = monthlyCost;
  }
  // restante da classe...
```

Com isso testado, movo o código comum para a superclasse. Pelo fato de esse código conter uma referência a um argumento do construtor, passo esse argumento como parâmetro.

class Party…

```
constructor(name){
  this._name = name;
}
```

class Employee…

```
constructor(name, id, monthlyCost) {
  super(name);
  this._id = id;
  this._monthlyCost = monthlyCost;
}
```

class Department…

```
constructor(name, staff){
  super(name);
  this._staff = staff;
}
```

Executo os testes, e pronto.

Na maioria das vezes, este será o procedimento para o construtor: cuide dos elementos comuns antes (com uma chamada a `super`), e então execute o trabalho extra necessário à subclasse. Ocasionalmente, porém, haverá algum

comportamento comum depois.

Considere o exemplo a seguir:

class Employee…

```
constructor (name) {...}
get isPrivileged() {...}
assignCar() {...}
```

class Manager extends Employee…

```
constructor(name, grade) {
  super(name);
  this._grade = grade;
  if (this.isPrivileged) this.assignCar(); // toda subclasse faz isso
}

get isPrivileged() {
  return this._grade > 4;
}
```

O problema, nesse exemplo, está no fato de que a chamada a isPrivileged não pode ser feita antes de o campo grade receber um valor, e isso só pode ser feito na subclasse.

Nesse caso, aplico *Extrair função (Extract Function)* no código comum:

class Manager…

```
constructor(name, grade) {
  super(name);
  this._grade = grade;
  this.finishConstruction();
}
finishConstruction() {
  if (this.isPrivileged) this.assignCar();
}
```

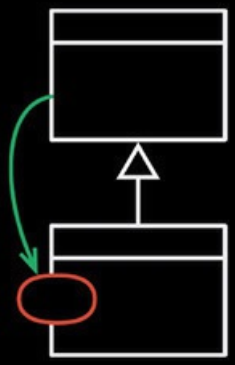
Em seguida, uso *Subir método (Pull Up Method)* para movê-lo para a superclasse.

class Employee…

```
finishConstruction() {
  if (this.isPrivileged) this.assignCar();
}
```

Descer método (Push Down Method)

inversa de: *Subir método (Pull Up Method)*



```
class Employee {
  get quota {...}
}

class Engineer extends Employee
{...}
class Salesman extends Employee
{...}
```

```
class Employee {...}
class Engineer extends Employee
{...}
class Salesman extends Employee
{
  get quota {...}
}
```

Motivação

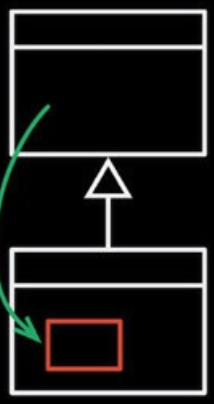
Se um método for relevante somente para uma subclasse (ou para uma pequena parcela das subclasses), removê-lo da superclasse e colocá-lo apenas na(s) subclasse(s) deixará esse código mais claro. Só será possível fazer essa refatoração se quem fizer a chamada souber que está trabalhando com uma subclasse em particular – caso contrário, será necessário usar *Substituir condicional por polimorfismo (Replace Conditional with Polymorphism)*, com algum comportamento placebo na superclasse.

Procedimento

- Copie o método para toda subclasse que precisar dele.
- Remova o método da superclasse.
- Teste.
- Remova o método de cada superclasse que não precisar dele.
- Teste.

Descer campo (Push Down Field)

inversa de: *Subir campo (Pull Up Field)*



```
class Employee { // Java
  private String quota;
}

class Engineer extends Employee
{...}
class Salesman extends Employee
{...}
```

```
class Employee {...}
class Engineer extends Employee
{...}

class Salesman extends Employee
{
  protected String quota;
}
```

Motivação

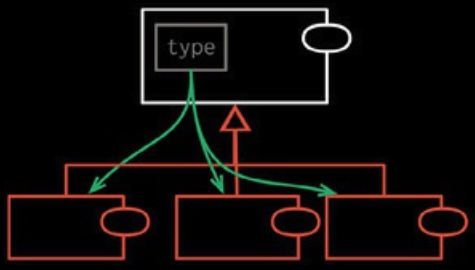
Se um campo for usado somente por uma subclasse (ou por uma pequena parcela das subclasses), posso mover esse campo para essa(s) subclasse(s).

Procedimento

- Declare o campo em todas as subclasses que precisarem dele.
- Remova o campo da superclasse.
- Teste.
- Remova o campo de todas as subclasses que não precisarem dele.
- Teste.

Substituir código de tipos por subclasses (Replace Type Code with Subclasses)

incorpora: *Substituir enumeração pelo padrão State/Strategy* (Replace Type Code with State/Strategy)
incorpora: *Extrair subclasse* (Extract Subclass)
inversa de: *Remover subclasse (Remove Subclass)*




```
function createEmployee(name,
type) {
  return new Employee(name, type);
}
```

```
function createEmployee(name, type) {
  switch (type) {
    case "engineer": return new
Engineer(name);
```

```
      case "salesman": return new
Salesman(name);
      case "manager": return new Manager
(name);
   }
```

Motivação

Sistemas de software muitas vezes precisam representar diferentes tipos de itens parecidos. Posso classificar funcionários (employees) de acordo com seu tipo de emprego (engenheiro [engineer], gerente [manager], vendedor [salesman]), ou pedidos conforme a sua prioridade (rápida [rush], comum [regular]). Minha primeira ferramenta para lidar com isso é ter algum tipo de campo com um código de tipo – dependendo da linguagem, pode ser um enumerado, um símbolo, uma string ou um número. Com frequência, esse código de tipo será proveniente de um serviço externo que disponibilize os dados com os quais estou trabalhando.

Na maioria das vezes, um código de tipo como esse é o necessário. No entanto, há duas situações em que eu poderia ter algo a mais, e esse algo a mais são as subclasses. Dois aspectos são particularmente atraentes no que diz respeito às subclasses. Em primeiro lugar, elas me permitem usar polimorfismo para lidar com lógicas condicionais. Acho isso muito conveniente quando tenho diversas funções que chamam diferentes comportamentos, de acordo com o valor do código de tipo. Com subclasses, posso aplicar *Substituir condicional por polimorfismo (Replace Conditional with Polymorphism)* nessas funções.

O segundo caso é aquele em que tenho campos ou métodos válidos somente para valores específicos de um código de tipo, por exemplo, uma quota de vendas que é aplicável somente ao código de tipo “salesman” (vendedor). Posso então criar a subclasse e aplicar *Descer campo (Push Down Field)*. Embora eu possa incluir uma lógica de validação para garantir que um campo seja usado somente quando o código de tipo tiver o valor correto, usar uma subclasse deixa o relacionamento mais explícito.

Ao usar *Substituir código de tipos por subclasses*, devo considerar se vou aplicá-la diretamente na classe que estou considerando, ou no próprio código de tipo. Faço com que engenheiro (engineer) seja um subtipo de funcionário (employee), ou devo criar uma propriedade para tipo de funcionário na classe funcionário, que poderá ter subtipos para engenheiro e para gerente

(manager)? Usar subclasses diretas é mais simples, mas não poderei usá-las para o tipo de trabalho caso eu precise delas para outras tarefas. Também não posso usar subclasses diretas se o tipo for mutável. Se for necessário mover as subclasses para uma propriedade associada ao tipo do funcionário, poderei fazer isso usando *Substituir primitivo por objeto (Replace Primitive with Object)* no código de tipo, de modo a criar uma classe para o tipo do funcionário, e então usar *Substituir código de tipos por subclasses* nessa nova classe.

Procedimento

- Autoencapsule o campo com o código de tipo.
- Escolha um valor do código de tipo. Crie uma subclasse para esse código. Sobrescreva o getter do código de tipo para que devolva o valor literal desse tipo.
- Crie uma lógica de seletor para fazer o mapeamento do parâmetro do código de tipo para a nova subclasse.

  Com uma herança direta, use *Substituir construtor por função de factory (Replace Constructor with Factory Function)* e coloque a lógica do seletor na factory. Com uma herança indireta, a lógica do seletor poderá permanecer no construtor.
- Teste.
- Repita a criação da subclasse e o acréscimo da lógica do seletor para cada valor do código de tipo. Teste após cada mudança.
- Remova o campo para o código de tipo.
- Teste.
- Use *Descer método (Push Down Method)* e *Substituir condicional por polimorfismo (Replace Conditional with Polymorphism)* em qualquer método que use os métodos de acesso ao código de tipo. Depois que tudo tiver sido substituído, você poderá remover esses métodos de acesso.

Exemplo

Começarei com o exemplo de funcionário (employee) a seguir:

class Employee…

```
constructor(name, type){
```

```
    this.validateType(type);
    this._name = name;
    this._type = type;
  }
  validateType(arg) {
    if (!["engineer", "manager", "salesman"].includes(arg))
      throw new Error(`Employee cannot be of type ${arg}`);
  }
  toString() {return `${this._name} (${this._type})`;}
```

Meu primeiro passo é usar *Encapsular variável (Encapsulate Variable)* para autoencapsular o código de tipo.

class Employee…

```
  get type() {return this._type;}
  toString() {return `${this._name} (${this.type})`;}
```

Observe que toString usa o novo getter, com a remoção do underscore.

Escolho um código de tipo, o código para engenheiro (engineer), para começar. Uso herança direta, criando uma subclasse da própria classe de funcionário. Essa subclasse de funcionário é simples – ela somente sobrescreve o getter do código de tipo com o valor literal apropriado.

```
class Engineer extends Employee {
    get type() {return "engineer";}
}
```

Embora construtores JavaScript possam devolver outros objetos, o código ficará confuso se eu tentar colocar aí a lógica do seletor, pois essa lógica ficará misturada com a inicialização do campo. Assim, uso *Substituir construtor por função de factory (Replace Constructor with Factory Function)* a fim de criar um novo espaço para ela.

```
function createEmployee(name, type) {
  return new Employee(name, type);
}
```

Para usar a nova subclasse, adiciono a lógica do seletor na factory.

```
function createEmployee(name, type) {
  switch (type) {
    case "engineer": return new Engineer(name, type);
  }
  return new Employee(name, type);
}
```

Testo para garantir que esse código funcione corretamente. Porém, como sou

paranoico, altero depois o valor de retorno da função que a classe engenheiro sobrescreve e testo de novo para garantir que o teste falhe. Desse modo, saberei que a subclasse está sendo usada. Corrijo o valor de retorno e prossigo com os demais casos. Posso trabalhar neles, um de cada vez, testando após cada mudança.

```
class Salesman extends Employee {
  get type() {return "salesman";}
}
class Manager extends Employee {
  get type() {return "manager";}
}
function createEmployee(name, type) {
  switch (type) {
    case "engineer": return new Engineer(name, type);
    case "salesman": return new Salesman(name, type);
    case "manager": return new Manager (name, type);
  }
  return new Employee(name, type);
}
```

Depois que tratar todos os casos, posso remover o campo com o código de tipo e o método de leitura da superclasse (os métodos que estão nas subclasses devem ser mantidos).

class Employee…

```
constructor(name, type){
  this.validateType(type);
  this._name = name;
  this._type = type;
}
get type() {return this._type;}
toString() {return `${this._name} (${this.type})`;}
```

Após testar e garantir que tudo continua bem, posso remover a lógica de validação, pois o switch tem efetivamente a mesma função.

class Employee…

```
  constructor(name, type){
    this.validateType(type);
    this._name = name;
  }
function createEmployee(name, type) {
  switch (type) {
    case "engineer": return new Engineer(name, type);
```

```
    case "salesman": return new Salesman(name, type);
    case "manager": return new Manager (name, type);
    default: throw new Error(`Employee cannot be of type ${type}`);
  }
  return new Employee(name, type);
}
```

O argumento de tipo do construtor agora é inútil, portanto será vítima de *Mudar declaração de função (Change Function Declaration)*.

class Employee…

```
  constructor(name, type){
    this._name = name;
  }
function createEmployee(name, type) {
  switch (type) {
    case "engineer": return new Engineer(name, type);
    case "salesman": return new Salesman(name, type);
    case "manager": return new Manager (name, type);
    default: throw new Error(`Employee cannot be of type ${type}`);
  }
}
```

Ainda tenho os métodos de acesso ao código de tipo nas subclasses – `get type`. Em geral, eu os removeria também, mas isso pode exigir um pouco mais de tempo por causa de outros métodos que dependam deles. Usarei *Substituir condicional por polimorfismo (Replace Conditional with Polymorphism)* e *Descer método (Push Down Method)* para cuidar deles. Em algum momento, não terei mais nenhum código que use os getters de `type`, portanto eu os sujeitarei à suave misericórdia de *Remover código morto (Remove Dead Code)*.

Exemplo: usando herança indireta

Vamos retornar ao caso inicial – dessa vez, porém, já tenho subclasses para funcionários que trabalham meio período (part-time employee) e em período integral (full-time employee), portanto não posso criar subclasses de `Employee` para os códigos de tipo. Outra razão para não usar herança direta está em preservar a capacidade de modificar o tipo do funcionário.

class Employee…

```
constructor(name, type){
  this.validateType(type);
```

```
    this._name = name;
    this._type = type;
  }
  validateType(arg) {
    if (!["engineer", "manager", "salesman"].includes(arg))
      throw new Error(`Employee cannot be of type ${arg}`);
  }
  get type() {return this._type;}
  set type(arg) {this._type = arg;}

  get capitalizedType() {
    return this._type.charAt(0).toUpperCase() + this._type.substr(1).toLowerCase();
  }
  toString() {
    return `${this._name} (${this.capitalizedType})`;
  }
```

Dessa vez, toString é um pouco mais complicado a fim de permitir que eu demonstre algo em breve.

Meu primeiro passo é usar *Substituir primitivo por objeto (Replace Primitive with Object)* no código de tipo.

```
class EmployeeType {
  constructor(aString) {
    this._value = aString;
  }
  toString() {return this._value;}
}
```

class Employee…

```
  constructor(name, type){
    this.validateType(type);
    this._name = name;
    this.type = type;
  }
  validateType(arg) {
    if (!["engineer", "manager", "salesman"].includes(arg))
      throw new Error(`Employee cannot be of type ${arg}`);
  }
  get typeString() {return this._type.toString();}
  get type() {return this._type;}
  set type(arg) {this._type = new EmployeeType(arg);}

  get capitalizedType() {
    return this.typeString.charAt(0).toUpperCase()
```

```
      + this.typeString.substr(1).toLowerCase();
  }

  toString() {
    return `${this._name} (${this.capitalizedType})`;
  }
```

Então aplico o procedimento usual de *Substituir código de tipos por subclasses* no tipo do funcionário.

class Employee…

```
    set type(arg) {this._type = Employee.createEmployeeType(arg);}
    static createEmployeeType(aString) {
      switch(aString) {
        case "engineer": return new Engineer();
        case "manager": return new Manager ();
        case "salesman": return new Salesman();
        default: throw new Error(`Employee cannot be of type ${aString}`);
      }
    }
  class EmployeeType {
  }
  class Engineer extends EmployeeType {
    toString() {return "engineer";}
  }
  class Manager extends EmployeeType {
    toString() {return "manager";}
  }
  class Salesman extends EmployeeType {
    toString() {return "salesman";}
  }
```

Se eu fosse deixar o código como está, poderia remover o EmployeeType vazio. No entanto, prefiro manter essa classe, pois ela deixa explícito o relacionamento entre as várias subclasses. Também é um lugar conveniente para mover outros comportamentos, por exemplo, a lógica de passar para letras maiúsculas (capitalization) que incluí no exemplo, especificamente para demonstrar essa questão.

class Employee…

```
  toString() {
    return `${this._name} (${this.type.capitalizedName})`;
  }
```

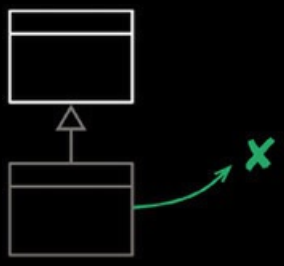
class EmployeeType…

```
get capitalizedName() {
  return this.toString().charAt(0).toUpperCase()
    + this.toString().substr(1).toLowerCase();
}
```

Para aqueles que conhecem a primeira edição do livro, esse exemplo essencialmente toma o lugar de *Substituir enumeração pelo padrão State/Strategy*. Penso agora naquela refatoração como *Substituir código de tipos por subclasses* usando herança indireta, portanto não considerei que valesse a pena criar uma entrada própria para ela no catálogo. (De qualquer maneira, jamais gostei do nome.)

Remover subclasse (Remove Subclass)

anteriormente: *Substituir subclasse por campos* (Replace Subclass with Fields)
inversa de: *Substituir código de tipos por subclasses (Replace Type Code with Subclasses)*



```
class Person {
  get genderCode() {return
"X";}
}
class Male extends Person {
  get genderCode() {return
"M";}
}
class Female extends Person {
  get genderCode() {return
"F";}
}
```

```
class Person {
```

```
  get genderCode() {return
this._genderCode;}
}
```

Motivação

As subclasses são úteis. Elas oferecem suporte para variações em estruturas de dados e para um comportamento polimórfico. São uma ótima maneira de programar com base na diferença. Contudo, um sistema de software evolui e as subclasses podem perder seu valor à medida que as variações tratadas por elas são movidas para outros lugares ou são totalmente removidas. Às vezes, subclasses são acrescentadas em antecipação a funcionalidades que jamais chegam a ser implementadas ou que acabam sendo construídas de modo que não precisam das subclasses.

Há um custo para compreender uma subclasse, e, se ela fizer muito pouco, não valerá mais a pena tê-la. Quando esse momento chegar, é melhor remover a subclasse, substituindo-a por um campo em sua superclasse.

Procedimento

- Use *Substituir construtor por função de factory (Replace Constructor with Factory Function)* no construtor da subclasse.

Se os clientes dos construtores usarem um campo de dado para decidir qual subclasse deve ser criada, coloque essa lógica de decisão em um método de factory da superclasse.

- Se houver algum código que teste os tipos das subclasses, use *Extrair função (Extract Function)* no teste de tipo, e *Mover função (Move Function)* para movê-lo para a superclasse. Teste após cada mudança.
- Crie um campo para representar o tipo da subclasse.
- Modifique os métodos que referenciam a subclasse de modo que usem o novo campo de tipo.
- Apague a subclasse.
- Teste.

Com frequência, essa refatoração é usada em um grupo de subclasses de uma só vez – nesse caso, execute os passos para encapsulá-las (adicionar função de factory, mover os testes de tipo) antes e, então, individualmente, insira o código na superclasse.

Exemplo

Começarei com estes trechos de subclasses:

class Person…

```
    constructor(name) {
      this._name = name;
    }
    get name() {return this._name;}
    get genderCode() {return "X";}
    // trecho omitido
  class Male extends Person {
    get genderCode() {return "M";}
  }

  class Female extends Person {
    get genderCode() {return "F";}
  }
```

Se isso for tudo que uma subclasse faz, não vale realmente a pena tê-la. Porém, antes de remover essas subclasses, em geral compensa verificar se há algum comportamento dependente de subclasse nos clientes, que deva ser movido para as subclasses. Nesse exemplo, não acho nada que justifique manter as subclasses.

cliente…

```
  const numberOfMales = people.filter(p => p instanceof Male).length;
```

Sempre que quero mudar o modo de representar algo, procuro primeiro encapsular a representação atual a fim de minimizar o impacto em qualquer código de cliente. Quando se trata de criar subclasses, faço o seu encapsulamento usando *Substituir construtor por função de factory (Replace Constructor with Factory Function)*. Nesse exemplo, há duas maneiras possíveis de criar a factory.

O modo mais direto é criar um método de factory para cada construtor.

```
  function createPerson(name) {
    return new Person(name);
  }
  function createMale(name) {
    return new Male(name);
  }
  function createFemale(name) {
    return new Female(name);
```

```
}
```

Entretanto, apesar de essa ser a opção direta, objetos como esses muitas vezes são carregados a partir de um código que usa diretamente o código de gênero (gender).

```
function loadFromInput(data) {
  const result = [];
  data.forEach(aRecord => {
    let p;
    switch (aRecord.gender) {
      case 'M': p = new Male(aRecord.name); break;
      case 'F': p = new Female(aRecord.name); break;
      default: p = new Person(aRecord.name);
    }
    result.push(p);
  });
  return result;
}
```

Nesse caso, acho melhor usar *Extrair função (Extract Function)* na lógica de seleção que determina a classe a ser criada e fazer dela a função de factory.

```
function createPerson(aRecord) {
  let p;
  switch (aRecord.gender) {
    case 'M': p = new Male(aRecord.name); break;
    case 'F': p = new Female(aRecord.name); break;
    default: p = new Person(aRecord.name);
  }
  return p;
}
function loadFromInput(data) {
  const result = [];
  data.forEach(aRecord => {
    result.push(createPerson(aRecord));
  });
  return result;
}
```

Enquanto faço isso, organizo as duas funções. Usarei *Internalizar variável (Inline Variable)* em createPerson:

```
function createPerson(aRecord) {
  switch (aRecord.gender) {
    case 'M': return new Male (aRecord.name);
    case 'F': return new Female(aRecord.name);
```

```
      default: return new Person(aRecord.name);
    }
  }
```

e *Substituir laço por pipeline (Replace Loop with Pipeline)* em loadFromInput:

```
function loadFromInput(data) {
  return data.map(aRecord => createPerson(aRecord));
}
```

A factory encapsula a criação das subclasses, mas há também o uso de instanceof – que nunca tem um cheiro bom. Uso *Extrair função (Extract Function)* na verificação do tipo.

cliente…

```
const numberOfMales = people.filter(p => isMale(p)).length;
function isMale(aPerson) {return aPerson instanceof Male;}
```

Em seguida, uso *Mover função (Move Function)* para movê-la para Person.

class Person…

```
get isMale() {return this instanceof Male;}
```

cliente…

```
const numberOfMales = people.filter(p => p.isMale).length;
```

Com essa refatoração feita, todo o conhecimento das subclasses agora está seguramente contido na superclasse e na função de factory. (Em geral, sou cauteloso com uma superclasse referenciando uma subclasse, mas esse código não vai durar até minha próxima xícara de chá, portanto não me preocuparei com isso.)

Agora adiciono um campo para representar a diferença entre as subclasses; como estou usando um código carregado a partir de outro lugar, posso simplesmente o usar também.

class Person…

```
constructor(name, genderCode) {
  this._name = name;
  this._genderCode = genderCode || "X";
}
get genderCode() {return this._genderCode;}
```

Na inicialização, defino o campo com o caso default. (Como uma observação adicional, embora a maioria das pessoas seja classificada como masculino (male) e feminino (female), há pessoas que não podem ser. Esquecer-se disso é um erro comum de modelagem.)

Então, tomo o caso masculino e insiro a sua lógica na superclasse. Isso envolve modificar a factory para que devolva uma Person e modificar qualquer teste com instanceof para que use o campo do código para gênero.

```
function createPerson(aRecord) {
  switch (aRecord.gender) {
    case 'M': return new Person(aRecord.name, "M");
    case 'F': return new Female(aRecord.name);
    default: return new Person(aRecord.name);
  }
}
```

class Person…

```
get isMale() {return "M" === this._genderCode;}
```

Testo, removo a subclasse para gênero masculino, testo novamente, e repito com a subclasse para o gênero feminino.

```
function createPerson(aRecord) {
  switch (aRecord.gender) {
    case 'M': return new Person(aRecord.name, "M");
    case 'F': return new Person(aRecord.name, "F");
    default: return new Person(aRecord.name);
  }
}
```

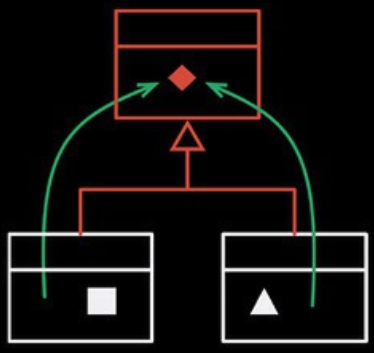
Acho irritante a falta de simetria no código de gênero. Um futuro leitor do código sempre se perguntará a respeito dessa falta de simetria. Assim, prefiro alterar o código para deixá-lo simétrico – se eu puder, faço isso sem introduzir nenhuma outra complexidade, caso desse exemplo.

```
function createPerson(aRecord) {
  switch (aRecord.gender) {
    case 'M': return new Person(aRecord.name, "M");
    case 'F': return new Person(aRecord.name, "F");
    default: return new Person(aRecord.name, "X");
  }
}
```

class Person…

```
constructor(name, genderCode) {
  this._name = name;
  this._genderCode = genderCode || "X";
}
```

Extrair superclasse (Extract Superclass)



```
class Department {
  get totalAnnualCost()
{...}
  get name() {...}
  get headCount() {...}
}

class Employee {
  get annualCost() {...}
  get name() {...}
  get id() {...}
}
```

```
class Party {
  get name() {...}
  get annualCost() {...}
}

class Department extends
Party {
  get annualCost() {...}
  get headCount() {...}
}

class Employee extends Party
{
  get annualCost() {...}
  get id() {...}
}
```

Motivação

Se eu vir duas classes fazendo tarefas parecidas, posso tirar proveito do mecanismo básico de herança para extrair suas semelhanças e inseri-las em uma superclasse. Posso usar *Subir campo (Pull Up Field)* para mover dados comuns para a superclasse, e *Subir método (Pull Up Method)* para mover comportamentos comuns.

Muitos que escrevem sobre orientação a objetos tratam a herança como algo que deve ser cuidadosamente planejado com antecedência, com base em algum tipo de estrutura de classificação do "mundo real". Estruturas de classificação como essas podem ser uma pista para usar herança – porém, com a mesma frequência, a herança é algo que percebo durante a evolução de um programa, à medida que identifico elementos comuns que eu gostaria de manter juntos.

Uma alternativa a *Extrair superclasse* é *Extrair classe (Extract Class)*. Nesse caso, você tem essencialmente uma escolha entre usar herança ou delegação como uma forma de unificar comportamentos duplicados. Em geral, *Extrair superclasse* é a abordagem mais simples, portanto farei isso antes, sabendo que poderei usar *Substituir superclasse por delegação (Replace Superclass with Delegate)* caso seja necessário no futuro.

Procedimento

- Crie uma superclasse vazia. Faça com que as classes originais sejam suas subclasses.

Se for necessário, use *Mudar declaração de função (Change Function Declaration)* nos construtores.

- Teste.
- Um a um, use *Subir corpo do construtor (Pull Up Constructor Body)*, *Subir método (Pull Up Method)* e *Subir campo (Pull Up Field)* a fim de mover elementos comuns para a superclasse.
- Analise os métodos restantes nas subclasses. Veja se há partes comuns. Se houver, use *Extrair função (Extract Function)*, seguida de *Subir método (Pull Up Method)*.
- Verifique os clientes das classes originais. Considere adaptá-los para que usem a interface da superclasse.

Exemplo

Considere as duas classes a seguir: elas compartilham algumas funcionalidades comuns – o nome e as noções de custos anuais (annual cost) e mensais (monthly cost):

```
class Employee {
  constructor(name, id, monthlyCost) {
    this._id = id;
    this._name = name;
    this._monthlyCost = monthlyCost;
  }
  get monthlyCost() {return this._monthlyCost;}
  get name() {return this._name;}
  get id() {return this._id;}

  get annualCost() {
    return this.monthlyCost * 12;
  }
}
class Department {
  constructor(name, staff){
    this._name = name;
    this._staff = staff;
  }
  get staff() {return this._staff.slice();}
  get name() {return this._name;}

  get totalMonthlyCost() {
    return this.staff
      .map(e => e.monthlyCost)
      .reduce((sum, cost) => sum + cost);
  }
  get headCount() {
    return this.staff.length;
  }
  get totalAnnualCost() {
    return this.totalMonthlyCost * 12;
  }
}
```

Posso deixar o comportamento comum mais explícito extraindo uma superclasse comum a partir dele.

Começo criando uma superclasse vazia e deixando que as duas classes estendam dessa superclasse.

```
class Party {}
class Employee extends Party {
  constructor(name, id, monthlyCost) {
    super();
    this._id = id;
    this._name = name;
    this._monthlyCost = monthlyCost;
  }
  // restante da classe...
class Department extends Party {
  constructor(name, staff){
    super();
    this._name = name;
    this._staff = staff;
  }
  // restante da classe...
```

Ao usar *Extrair superclasse*, gosto de começar pelos dados; em JavaScript, isso envolve manipular o construtor. Desse modo, começo com *Subir campo (Pull Up Field)* para subir o nome.

class Party…

```
constructor(name){
  this._name = name;
}
```

class Employee…

```
constructor(name, id, monthlyCost) {
  super(name);
  this._id = id;
  this._monthlyCost = monthlyCost;
}
```

class Department…

```
constructor(name, staff){
  super(name);
  this._staff = staff;
}
```

À medida que subo os dados para a superclasse, também posso aplicar *Subir método (Pull Up Method)* nos métodos associados. Em primeiro lugar, o nome:

class Party…

```
get name() {return this._name;}
```

class Employee…

```
get name() {return this._name;}
```

class Department…

```
get name() {return this._name;}
```

Tenho dois métodos com corpos parecidos.

class Employee…

```
get annualCost() {
  return this.monthlyCost * 12;
}
```

class Department…

```
get totalAnnualCost() {
  return this.totalMonthlyCost * 12;
}
```

Os métodos que eles usam, monthlyCost e totalMonthlyCost, têm nomes e corpos diferentes – mas eles representam o mesmo propósito? Em caso afirmativo, uso *Mudar declaração de função (Change Function Declaration)* para unificar seus nomes.

class Department…

```
get totalAnnualCost() {
  return this.monthlyCost * 12;
}
get monthlyCost() { … }
```

Em seguida, renomeio, de modo semelhante, para o caso dos custos anuais:

class Department…

```
get annualCost() {
  return this.monthlyCost * 12;
}
```

Agora posso aplicar *Subir método (Pull Up Method)* nos métodos para os custos anuais.

class Party…

```
get annualCost() {
  return this.monthlyCost * 12;
}
```

class Employee…

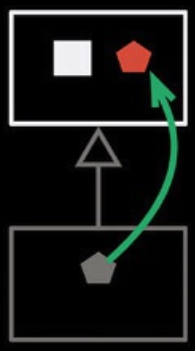
```
get annualCost() {
```

```
  return this.monthlyCost * 12;
}
```

class Department…

```
get annualCost() {
  return this.monthlyCost * 12;
}
```

Condensar Hierarquia (Collapse Hierarchy)



```
class Employee {...}
class Salesman extends Employee
{...}
```

```
class Employee
{...}
```

Motivação

Quando estiver refatorando uma hierarquia de classes, com frequência subo e desço recursos. À medida que a hierarquia evolui, às vezes percebo que uma classe e sua classe-pai já não são mais tão diferentes para justificar mantê-las separadas. Nesse ponto, eu as combinarei.

Procedimento

- Escolha a classe a ser removida.

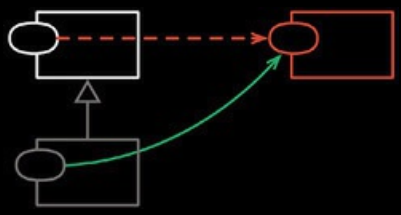
  Realizo a escolha com base no nome que fizer mais sentido no futuro. Se nenhum dos nomes for o melhor, escolho um arbitrariamente.

- Use *Subir campo (Pull Up Field)*, *Descer campo (Push Down Field)*,

Subir método (Pull Up Method) e *Descer método (Push Down Method)* a fim de mover todos os elementos para uma única classe.

- Faça adaptações em qualquer referência à vítima para mudá-las de modo que usem a classe que permanecerá.
- Remova a classe vazia.
- Teste.

Substituir subclasse por delegação (Replace Subclass with Delegate)



```
class Order {
  get daysToShip() {
    return
this._warehouse.daysToShip;
  }
}

class PriorityOrder extends Order {
  get daysToShip() {
    return
this._priorityPlan.daysToShip;
  }
}
```

```
class Order {
  get daysToShip() {
    return (this._priorityDelegate)
      ?
this._priorityDelegate.daysToShip
      : this._warehouse.daysToShip;
  }
}
```

```
class PriorityOrderDelegate {
  get daysToShip() {
    return
this._priorityPlan.daysToShip
  }
}
```

Motivação

Se houver alguns objetos cujos comportamentos variem conforme a categoria do objeto, o mecanismo natural para expressar isso será a herança. Coloco todos os dados e comportamentos comuns na superclasse e deixo que cada subclasse adicione e sobrescreva os recursos à medida que for necessário. Linguagens orientadas a objetos simplificam essa implementação e, portanto, esse é um mecanismo conhecido.

Apesar disso, a herança tem suas desvantagens. A mais óbvia é que ela é uma cartada que só pode ser usada uma única vez. Se houver mais de um motivo para variar algo, a herança poderá ser usada somente em um único eixo de variação. Desse modo, se eu quiser variar o comportamento das pessoas de acordo com a categoria idade e o seu nível de renda, poderei ter subclasses para jovens e idosos, ou para ricos e pobres – mas não para ambos.

Outro problema é que a herança introduz um relacionamento muito íntimo entre as classes. Qualquer mudança que eu quiser fazer na classe-pai poderá facilmente causar erros nas classes-filhas, portanto tenho de ser cuidadoso e entender como as filhas derivam da superclasse. Esse problema piora quando a lógica das duas classes estiver em módulos distintos e houver equipes diferentes responsáveis por esses módulos.

A delegação cuida desses dois problemas. Posso delegar para várias classes diferentes por diferentes motivos. A delegação é um relacionamento comum entre objetos – assim, é possível ter uma interface clara com a qual trabalhar, o que significa muito menos acoplamento do que haveria com subclasses. Desse modo, é comum deparar com os problemas consequentes do uso de subclasses e aplicar *Substituir subclasse por delegação*.

Há um princípio conhecido que diz: “Prefira composição de objetos à herança de classe” (em que composição é efetivamente o mesmo que delegação). Muitas pessoas entendem isso como “herança deve ser considerada prejudicial”, e argumentam que a herança não deve ser jamais usada. Eu uso herança frequentemente, em parte porque sei que sempre posso

usar *Substituir subclasse por delegação* caso tenha de fazer uma mudança depois. A herança é um mecanismo importante, e é apropriada à tarefa na maioria das vezes, sem que apresente problemas. Assim, lanço mão dela antes, e passo para a delegação se ela começar a mostrar problemas. Esse uso, na verdade, é consistente com o princípio – extraído do livro da Gangue dos Quatro (Gang of Four) [gof], que explica como a herança e a composição funcionam juntas. O princípio foi uma reação ao uso excessivo da herança.

Aqueles que têm familiaridade com o livro da Gangue dos Quatro podem achar conveniente pensar nessa refatoração como uma substituição de subclasses pelos padrões State (Estado) e Strategy (Estratégia). Esses dois padrões são estruturalmente iguais, contando com um host fazendo a delegação para uma hierarquia separada. Nem todos os casos de *Substituir subclasse por delegação* envolvem uma hierarquia de herança para a delegação (como mostra o primeiro exemplo a seguir), mas definir uma hierarquia para os estados ou as estratégias muitas vezes será conveniente.

Procedimento

- Se houver muitas chamadas aos construtores, aplique *Substituir construtor por função de factory (Replace Constructor with Factory Function)*.
- Crie uma classe vazia para a delegação. Seu construtor deve aceitar qualquer dado específico de subclasse, assim como, em geral, uma referência para trás, para a superclasse.
- Acrescente um campo na superclasse para armazenar a classe de delegação.
- Modifique a criação da subclasse de modo que ela inicialize o campo de delegação com uma instância da classe de delegação.

Isso pode ser feito na função de factory, ou no construtor se ele tiver certeza de que a classe correta de delegação pode ser criada.

- Escolha um método da subclasse para mover para a classe de delegação.
- Use *Mover função (Move Function)* a fim de movê-lo para a classe de delegação. Não remova o código de delegação original.

Se o método precisar de elementos que devam ser movidos para a classe de delegação, mova-os. Se ele precisar de elementos que devam permanecer na superclasse, acrescente um campo na classe de delegação que faça referência à superclasse.

- Se o método original tiver chamadas fora da classe, mova o código de delegação original da subclasse para a superclasse, protegendo-o com uma verificação da presença da delegação. Se não tiver, aplique *Remover código morto (Remove Dead Code)*.

Se houver mais de uma subclasse, e você começar a duplicar código, use *Extrair superclasse (Extract Superclass)*. Nesse caso, qualquer método de delegação na superclasse original não precisará mais de proteção se o comportamento padrão for movido para a superclasse de delegação.

- Teste.
- Repita até que todos os métodos da subclasse tenham sido movidos.
- Encontre todas as chamadas de construtor das subclasses e modifique-as de modo que usem o construtor da superclasse.
- Teste.
- Use *Remover código morto (Remove Dead Code)* na subclasse.

Exemplo

Tenho uma classe que faz uma reserva (booking) para um show.

class Booking…

```
constructor(show, date) {
  this._show = show;
  this._date = date;
}
```

Há uma subclasse para uma reserva premium que leva em consideração vários extras disponíveis.

class PremiumBooking extends Booking…

```
constructor(show, date, extras) {
  super(show, date);
  this._extras = extras;
}
```

A reserva premium faz muitas mudanças naquilo que ela herda da superclasse. Como é típico nesse tipo de programação por diferença, em alguns casos a subclasse sobrescreve métodos da superclasse e, em outros, ela acrescenta novos métodos que são relevantes somente para a subclasse. Não descreverei todos eles, mas selecionarei alguns casos interessantes.

Inicialmente, temos uma sobrescrita simples. Reservas comuns oferecem

uma sessão de conversa (talkback) após o show, mas somente em dias que não sejam de pico.

class Booking…

```
get hasTalkback() {
  return this._show.hasOwnProperty('talkback') && !this.isPeakDay;
}
```

Reservas premium sobrescrevem isso e oferecem sessões de conversa todos os dias.

class PremiumBooking…

```
get hasTalkback() {
  return this._show.hasOwnProperty('talkback');
}
```

A determinação do preço é feita com uma sobrescrita semelhante, com uma pequena variação: os métodos da classe premium chamam o método da superclasse.

class Booking…

```
get basePrice() {
  let result = this._show.price;
  if (this.isPeakDay) result += Math.round(result * 0.15);
  return result;
}
```

class PremiumBooking…

```
get basePrice() {
  return Math.round(super.basePrice + this._extras.premiumFee);
}
```

No último exemplo, a reserva premium oferece um comportamento que não está presente na superclasse.

class PremiumBooking…

```
get hasDinner() {
  return this._extras.hasOwnProperty('dinner') && !this.isPeakDay;
}
```

A herança funciona bem nesse exemplo. Sou capaz de entender a classe-base sem ter de compreender a subclasse. A subclasse é definida apenas dizendo como ela difere do caso de base – tanto reduzindo a duplicação quanto informando claramente quais são as diferenças que ela introduz.

Na verdade, nem tudo é tão perfeito quanto implica o parágrafo anterior. Há

elementos na estrutura da superclasse que só fazem sentido por causa da subclasse – como métodos que foram fatorados de modo que facilitassem sobrescrever exatamente os tipos corretos de comportamento. Então, embora na maioria das vezes eu possa modificar a classe-base sem ter de entender as subclasses, há ocasiões em que uma ignorância deliberada como essa das subclasses pode fazer com que eu cause falhas em uma subclasse ao modificar a superclasse. No entanto, se essas ocasiões não forem tão comuns, compensa usar a herança – desde que eu tenha bons testes para detectar falhas em uma subclasse.

Então por que eu iria querer mudar uma situação satisfatória como essa usando *Substituir subclasse por delegação*? A herança é uma ferramenta que só pode ser usada uma vez – portanto, se eu tiver outro motivo para usá-la, e achar que me traria mais vantagens do que ter uma subclasse para reserva premium, terei de lidar com as reservas premium de outra maneira. Além do mais, talvez eu precise mudar dinamicamente da reserva default para a reserva premium – isto é, oferecer suporte para um método como aBooking.bePremium(). Em alguns casos, posso evitar isso criando um objeto totalmente novo (um exemplo comum é quando uma requisição HTTP carrega novos dados do servidor). Outras vezes, porém, preciso modificar uma estrutura de dados sem reconstruí-la do zero, e é difícil simplesmente substituir uma única reserva referenciada em vários lugares diferentes. Em situações como essa, talvez seja conveniente permitir que uma reserva mude de default para premium e vice-versa.

Quando essas necessidades surgirem, será necessário aplicar *Substituir subclasse por delegação*. Tenho clientes que chamam os construtores das duas classes para fazer as reservas:

cliente de reserva (booking)

```
aBooking = new Booking(show,date);
```

cliente de reserva premium

```
aBooking = new PremiumBooking(show, date, extras);
```

Remover subclasses alterará tudo isso, portanto prefiro encapsular as chamadas de construtor usando *Substituir construtor por função de factory (Replace Constructor with Factory Function)*.

nível mais alto…

```
function createBooking(show, date) {
```

```
  return new Booking(show, date);
}
function createPremiumBooking(show, date, extras) {
  return new PremiumBooking (show, date, extras);
}
```

cliente de reserva (booking)

```
aBooking = createBooking(show, date);
```

cliente de reserva premium

```
aBooking = createPremiumBooking(show, date, extras);
```

Agora crio a nova classe de delegação. Os parâmetros de seu construtor são aqueles usados somente na subclasse, junto com uma referência para trás, para o objeto de reserva. Precisarei dele porque vários métodos da subclasse exigem acesso a dados armazenados na superclasse. A herança facilita fazer isso; porém, com uma delegação, preciso de uma referência para trás.

class PremiumBookingDelegate…

```
constructor(hostBooking, extras) {
  this._host = hostBooking;
  this._extras = extras;
}
```

Agora faço a associação entre a nova classe de delegação com o objeto de reserva. Faço isso modificando a função de factory para reservas premium.

nível mais alto…

```
function createPremiumBooking(show, date, extras) {
  const result = new PremiumBooking (show, date, extras);
  result._bePremium(extras);
  return result;
}
```

class Booking…

```
_bePremium(extras) {
  this._premiumDelegate = new PremiumBookingDelegate(this, extras);
}
```

Uso um underscore na frente de `_bePremium` para sinalizar que ele não deve fazer parte da interface pública de `Booking`. É claro que, se o sentido de fazer essa refatoração é permitir que uma reserva mude para premium, o método poderá ser público.

De modo alternativo, posso fazer todas as associações no construtor de `Booking`. Para isso, é necessário ter alguma forma de sinalizar ao construtor que

temos uma reserva premium. Poderia ser um parâmetro extra ou simplesmente o uso de extras se eu puder ter certeza de que ele estará sempre presente quando usado com uma reserva premium. Nesse caso, prefiro ser explícito e fazer isso por meio da função de factory.

Com as estruturas definidas, é hora de começar a mover os comportamentos. O primeiro caso que considerarei é a sobrescrita simples de hasTalkback. Eis o código existente:

class Booking…

```
get hasTalkback() {
  return this._show.hasOwnProperty('talkback') && !this.isPeakDay;
}
```

class PremiumBooking…

```
get hasTalkback() {
  return this._show.hasOwnProperty('talkback');
}
```

Uso *Mover função (Move Function)* para mover o método da subclasse para a classe de delegação. Para que ele esteja bem acomodado em seu novo lar, encaminho qualquer acesso aos dados da superclasse com uma chamada a _host.

class PremiumBookingDelegate…

```
get hasTalkback() {
  return this._host._show.hasOwnProperty('talkback');
}
```

class PremiumBooking…

```
get hasTalkback() {
  return this._premiumDelegate.hasTalkback;
}
```

Testo para garantir que tudo esteja funcionando, e então apago o método da subclasse:

class PremiumBooking…

```
~~get hasTalkback() {~~
~~  return this._premiumDelegate.hasTalkback;~~
~~}~~
```

Executo os testes nesse ponto, esperando que alguns falhem.

Agora termino a mudança adicionando a lógica para despachar o código correto no método da superclasse de modo a usar a delegação, se ela estiver

presente.

class Booking…

```
get hasTalkback() {
  return (this._premiumDelegate)
    ? this._premiumDelegate.hasTalkback
    : this._show.hasOwnProperty('talkback') && !this.isPeakDay;
}
```

O próximo caso que veremos refere-se ao preço básico (base price).

class Booking…

```
get basePrice() {
  let result = this._show.price;
  if (this.isPeakDay) result += Math.round(result * 0.15);
  return result;
}
```

class PremiumBooking…

```
get basePrice() {
  return Math.round(super.basePrice + this._extras.premiumFee);
}
```

É quase o mesmo caso, porém há um pequeno detalhe na forma da chamada inconveniente a `super` (que é muito comum nesses tipos de casos de extensão com subclasse). Ao mover o código da subclasse para a classe de delegação, será necessário chamar o caso pai – mas não posso apenas chamar `this._host._basePrice` sem cair em uma recursão infinita.

Nesse caso, tenho duas opções. Uma delas consiste em aplicar *Extrair função (Extract Function)* no cálculo básico para permitir que eu separe a lógica a fim de despachar o código correto e o cálculo do preço. (O restante da modificação é igual ao caso anterior.)

class Booking…

```
get basePrice() {
  return (this._premiumDelegate)
    ? this._premiumDelegate.basePrice
    : this._privateBasePrice;
}

get _privateBasePrice() {
  let result = this._show.price;
  if (this.isPeakDay) result += Math.round(result * 0.15);
  return result;
```

```
  }
```

class PremiumBookingDelegate…

```
  get basePrice() {
    return Math.round(this._host._privateBasePrice + this._extras.premiumFee);
  }
```

De modo alternativo, posso reconfigurar o método da classe de delegação como uma extensão do método do preço base.

class Booking…

```
  get basePrice() {
    let result = this._show.price;
    if (this.isPeakDay) result += Math.round(result * 0.15);
    return (this._premiumDelegate)
      ? this._premiumDelegate.extendBasePrice(result)
      : result;
  }
```

class PremiumBookingDelegate…

```
  extendBasePrice(base) {
    return Math.round(base + this._extras.premiumFee);
  }
```

As duas soluções são razoáveis nesse exemplo; tenho uma pequena preferência pela última, pois é um pouco mais concisa.

O último caso é aquele em que há um método existente apenas na subclasse.

class PremiumBooking…

```
  get hasDinner() {
    return this._extras.hasOwnProperty('dinner') && !this.isPeakDay;
  }
```

Movo esse método da subclasse para a classe de delegação:

class PremiumBookingDelegate…

```
  get hasDinner() {
    return this._extras.hasOwnProperty('dinner') && !this._host.isPeakDay;
  }
```

Em seguida, acrescento a lógica para despachar o código correto em Booking:

class Booking…

```
  get hasDinner() {
    return (this._premiumDelegate)
      ? this._premiumDelegate.hasDinner
      : undefined;
```

```
}
```

Em JavaScript, acessar uma propriedade em um objeto no qual ela não está definida devolve `undefined`, portanto faço isso nesse caso. (Embora meu instinto diga para gerar um erro, que seria o caso em outras linguagens dinâmicas orientadas a objetos com as quais estou acostumado.)

Depois de ter removido todo o comportamento da subclasse, posso modificar o método de factory para que devolva a superclasse – e, depois de ter executado os testes para garantir que tudo está bem, apago a subclasse.

nível mais alto…

```
function createPremiumBooking(show, date, extras) {
  const result = new ~~Premium~~Booking (show, date, extras);
  result._bePremium(extras);
  return result;
}
~~class PremiumBooking extends Booking ...~~
```

Essa é uma das refatorações em que não acho que a refatoração sozinha melhore o código. A herança lida muito bem com essa situação, enquanto usar delegação envolve acrescentar a lógica para despachar o código, referências bidirecionais e, portanto, uma complexidade extra. Talvez ainda valha a pena fazer a refatoração, pois a vantagem de ter um status premium mutável ou a necessidade de usar a herança para outros propósitos podem compensar a desvantagem de perder a herança.

Exemplo: substituindo uma hierarquia

O exemplo anterior mostrou o uso de *Substituir subclasse por delegação* em uma única subclasse, mas posso fazer o mesmo com uma hierarquia completa.

```
function createBird(data) {
  switch (data.type) {
    case 'EuropeanSwallow':
      return new EuropeanSwallow(data);
    case 'AfricanSwallow':
      return new AfricanSwallow(data);
    case 'NorweigianBlueParrot':
      return new NorwegianBlueParrot(data);
    default:
      return new Bird(data);
  }
```

```
}
class Bird {
  constructor(data) {
    this._name = data.name;
    this._plumage = data.plumage;
  }
  get name() {return this._name;}

  get plumage() {
    return this._plumage || "average";
  }
  get airSpeedVelocity() {return null;}
}

class EuropeanSwallow extends Bird {
  get airSpeedVelocity() {return 35;}
}

class AfricanSwallow extends Bird {
  constructor(data) {
    super (data);
    this._numberOfCoconuts = data.numberOfCoconuts;
  }
  get airSpeedVelocity() {
    return 40 - 2 * this._numberOfCoconuts;
  }
}

class NorwegianBlueParrot extends Bird {
  constructor(data) {
    super (data);
    this._voltage = data.voltage;
    this._isNailed = data.isNailed;
  }

  get plumage() {
    if (this._voltage > 100) return "scorched";
    else return this._plumage || "beautiful";
  }

  get airSpeedVelocity() {
    return (this._isNailed) ? 0 : 10 + this._voltage / 10;
  }
}
```

O sistema em breve fará uma grande diferenciação entre pássaros silvestres (wild birds) e pássaros em cativeiro (captive birds). Essa diferença poderia

ser modelada na forma de duas subclasses de Bird: WildBird e CaptiveBird. No entanto, só posso usar a herança uma vez, portanto, se eu quiser usar subclasses para selvagem (wild) *versus* cativeiro (captive), será necessário deixar de usá-las para as espécies.

Se houver várias subclasses envolvidas, cuidarei delas, uma de cada vez, começando pela mais simples – nesse caso, com EuropeanSwallow. Crio uma classe vazia para a delegação.

```
class EuropeanSwallowDelegate {
}
```

Não incluo ainda nenhum parâmetro de dados ou de referência para trás. Nesse exemplo, eu os introduzirei conforme forem necessários.

Tenho de decidir em que lugar cuidarei da inicialização do campo de delegação. No exemplo, como tenho todas as informações no argumento de dados único para o construtor, decidi fazer isso no construtor. Como há várias classes de delegação que eu poderia acrescentar, crio uma função para selecionar a classe de delegação correta com base no código de tipo presente nos dados.

class Bird…

```
constructor(data) {
  this._name = data.name;
  this._plumage = data.plumage;
  this._speciesDelegate = this.selectSpeciesDelegate(data);
}
selectSpeciesDelegate(data) {
  switch(data.type) {
    case 'EuropeanSwallow':
      return new EuropeanSwallowDelegate();
    default: return null;
  }
}
```

Agora que tenho a estrutura definida, posso aplicar *Mover função (Move Function)* na velocidade de voo do European Swallow (andorinha europeia).

class EuropeanSwallowDelegate…

```
get airSpeedVelocity() {return 35;}
```

class EuropeanSwallow…

```
get airSpeedVelocity() {return this._speciesDelegate.airSpeedVelocity;}
```

Modifico airSpeedVelocity na superclasse para que chame uma delegação, se

houver.

class Bird…

```
get airSpeedVelocity() {
  return this._speciesDelegate ? this._speciesDelegate.airSpeedVelocity : null;
}
```

Removo a subclasse.

```
class EuropeanSwallow extends Bird {
  get airSpeedVelocity() {return this._speciesDelegate.airSpeedVelocity;}
}
```

nível mais alto…

```
function createBird(data) {
  switch (data.type) {
    case 'EuropeanSwallow':
      return new EuropeanSwallow(data);
    case 'AfricanSwallow':
      return new AfricanSwallow(data);
    case 'NorweigianBlueParrot':
      return new NorwegianBlueParrot(data);
    default:
      return new Bird(data);
  }
}
```

Em seguida, cuidarei do African Swallow (andorinha africana). Crio uma classe; dessa vez, o construtor precisa dos dados.

class AfricanSwallowDelegate…

```
constructor(data) {
  this._numberOfCoconuts = data.numberOfCoconuts;
}
```

class Bird…

```
selectSpeciesDelegate(data) {
  switch(data.type) {
    case 'EuropeanSwallow':
      return new EuropeanSwallowDelegate();
    case 'AfricanSwallow':
      return new AfricanSwallowDelegate(data);
    default: return null;
  }
}
```

Uso *Mover função (Move Function)* em airSpeedVelocity.

class AfricanSwallowDelegate…

```
  get airSpeedVelocity() {
    return 40 - 2 * this._numberOfCoconuts;
  }
```

class AfricanSwallow…

```
  get airSpeedVelocity() {
    return this._speciesDelegate.airSpeedVelocity;
  }
```

Posso agora remover a subclasse para o African Swallow.

```
  class AfricanSwallow extends Bird {
    // todo o corpo ...
    }
  function createBird(data) {
    switch (data.type) {
      case 'AfricanSwallow':
        return new AfricanSwallow(data);
      case 'NorweigianBlueParrot':
        return new NorwegianBlueParrot(data);
      default:
        return new Bird(data);
    }
  }
```

Temos agora o Norwegian Blue (azul norueguês). Criar a classe e mover o método para velocidade de voo exigem os mesmos passos anteriores, portanto simplesmente mostrarei o resultado.

class Bird…

```
  selectSpeciesDelegate(data) {
    switch(data.type) {
      case 'EuropeanSwallow':
        return new EuropeanSwallowDelegate();
      case 'AfricanSwallow':
        return new AfricanSwallowDelegate(data);
      case 'NorweigianBlueParrot':
        return new NorwegianBlueParrotDelegate(data);
      default: return null;
    }
  }
```

class NorwegianBlueParrotDelegate…

```
  constructor(data) {
```

```
    this._voltage = data.voltage;
    this._isNailed = data.isNailed;
  }
  get airSpeedVelocity() {
    return (this._isNailed) ? 0 : 10 + this._voltage / 10;
  }
```

Tudo vai bem até agora, mas o Norwegian Blue sobrescreve a propriedade `plumage`, e não tive de lidar com isso nos demais casos. O *Mover função (Move Function)* inicial é simples, mas exige que o construtor seja modificado a fim de inserir uma referência para trás para Bird.

class NorwegianBlueParrot…

```
  get plumage() {
    return this._speciesDelegate.plumage;
  }
```

class NorwegianBlueParrotDelegate…

```
  get plumage() {
    if (this._voltage > 100) return "scorched";
    else return this._bird._plumage || "beautiful";
  }
  constructor(data, bird) {
    this._bird = bird;
    this._voltage = data.voltage;
    this._isNailed = data.isNailed;
  }
```

class Bird…

```
  selectSpeciesDelegate(data) {
    switch(data.type) {
      case 'EuropeanSwallow':
        return new EuropeanSwallowDelegate();
      case 'AfricanSwallow':
        return new AfricanSwallowDelegate(data);
      case 'NorweigianBlueParrot':
        return new NorwegianBlueParrotDelegate(data, this);
      default: return null;
    }
  }
```

O passo complicado é a remoção do método `plumage` da subclasse. Se eu fizer:

class Bird…

```
  get plumage() {
```

```
    if (this._speciesDelegate)
      return this._speciesDelegate.plumage;
    else
      return this._plumage || "average";
  }
```

verei um bocado de erros porque não há nenhuma propriedade para plumagem nas classes de delegação das demais espécies.

Eu poderia usar uma condicional mais precisa:

class Bird…

```
  get plumage() {
    if (this._speciesDelegate instanceof NorwegianBlueParrotDelegate)
      return this._speciesDelegate.plumage;
    else
      return this._plumage || "average";
  }
```

No entanto espero que, tanto quanto eu, você perceba um mau cheiro tão ruim quanto o de um papagaio (parrot) em decomposição. Usar uma verificação explícita de classe como essa raramente é uma boa ideia.

Outra opção é implementar o caso default nas outras delegações.

class Bird…

```
  get plumage() {
    if (this._speciesDelegate)
      return this._speciesDelegate.plumage;
    else
      return this._plumage || "average";
  }
```

class EuropeanSwallowDelegate…

```
  get plumage() {
    return this._bird._plumage || "average";
  }
```

class AfricanSwallowDelegate…

```
  get plumage() {
    return this._bird._plumage || "average";
  }
```

Porém, o método default para plumagem ficará duplicado. Como se isso não fosse suficientemente ruim, também terei algumas duplicações extras nos construtores para atribuir a referência para trás.

A solução para a duplicação, naturalmente, é a herança – aplico *Extrair*

superclasse (Extract Superclass) nas delegações das espécies:

```
class SpeciesDelegate {
  constructor(data, bird) {
    this._bird = bird;
  }
  get plumage() {
    return this._bird._plumage || "average";
  }
class EuropeanSwallowDelegate extends SpeciesDelegate {
class AfricanSwallowDelegate extends SpeciesDelegate {
  constructor(data, bird) {
    super(data,bird);
    this._numberOfCoconuts = data.numberOfCoconuts;
  }
class NorwegianBlueParrotDelegate extends SpeciesDelegate {
  constructor(data, bird) {
    super(data, bird);
    this._voltage = data.voltage;
    this._isNailed = data.isNailed;
  }
```

Com efeito, agora que tenho uma superclasse, posso mover qualquer comportamento default de Bird para SpeciesDelegate, garantindo que sempre haja algo no campo speciesDelegate.

class Bird…

```
selectSpeciesDelegate(data) {
  switch(data.type) {
    case 'EuropeanSwallow':
      return new EuropeanSwallowDelegate(data, this);
    case 'AfricanSwallow':
      return new AfricanSwallowDelegate(data, this);
    case 'NorweigianBlueParrot':
      return new NorwegianBlueParrotDelegate(data, this);
    default: return new SpeciesDelegate(data, this);
  }
}
// restante do código de Bird...
get plumage() {return this._speciesDelegate.plumage;}
get airSpeedVelocity() {return this._speciesDelegate.airSpeedVelocity;}
```

class SpeciesDelegate…

```
get airSpeedVelocity() {return null;}
```

Gosto dessa solução, pois ela simplifica os métodos de delegação em Bird. Posso ver facilmente qual comportamento está sendo delegado para as classes de delegação das espécies e quais devem ficar para trás.

Eis o estado final dessas classes:

```
function createBird(data) {
  return new Bird(data);
}
class Bird {
  constructor(data) {
    this._name = data.name;
    this._plumage = data.plumage;
    this._speciesDelegate = this.selectSpeciesDelegate(data);
  }
  get name() {return this._name;}
  get plumage() {return this._speciesDelegate.plumage;}
  get airSpeedVelocity() {return this._speciesDelegate.airSpeedVelocity;}

  selectSpeciesDelegate(data) {
    switch(data.type) {
      case 'EuropeanSwallow':
        return new EuropeanSwallowDelegate(data, this);
      case 'AfricanSwallow':
        return new AfricanSwallowDelegate(data, this);
      case 'NorweigianBlueParrot':
        return new NorwegianBlueParrotDelegate(data, this);
      default: return new SpeciesDelegate(data, this);
    }
  }
  // restante do código de Bird...
}

class SpeciesDelegate {
  constructor(data, bird) {
    this._bird = bird;
  }
  get plumage() {
    return this._bird._plumage || "average";
  }
  get airSpeedVelocity() {return null;}
}

class EuropeanSwallowDelegate extends SpeciesDelegate {
  get airSpeedVelocity() {return 35;}
}
```

```
class AfricanSwallowDelegate extends SpeciesDelegate {
  constructor(data, bird) {
    super(data,bird);
    this._numberOfCoconuts = data.numberOfCoconuts;
  }
  get airSpeedVelocity() {
    return 40 - 2 * this._numberOfCoconuts;
  }
}

class NorwegianBlueParrotDelegate extends SpeciesDelegate {
  constructor(data, bird) {
    super(data, bird);
    this._voltage = data.voltage;
    this._isNailed = data.isNailed;
  }
  get airSpeedVelocity() {
    return (this._isNailed) ? 0 : 10 + this._voltage / 10;
  }
  get plumage() {
    if (this._voltage > 100) return "scorched";
    else return this._bird._plumage || "beautiful";
  }
}
```

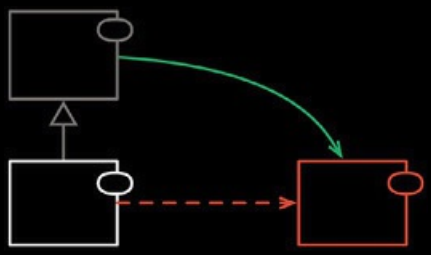
Esse exemplo substitui as subclasses originais por uma delegação, mas há ainda uma estrutura de herança muito parecida em SpeciesDelegate. Eu ganhei algo com essa refatoração, além de me livrar da herança em Bird? A herança de espécies tem um escopo mais claro agora, incluindo somente os dados e as funções que variam de acordo com as espécies. Qualquer código que seja igual para todas as espécies permanece em Bird e em suas futuras subclasses.

Eu poderia aplicar a mesma ideia de criar uma delegação na superclasse no exemplo anterior de reservas. Isso me permitiria substituir aqueles métodos em Booking contendo a lógica para despachar o código correto por chamadas simples de delegação, deixando que a herança resolvesse como despachar a lógica. No entanto, está quase na hora do jantar, portanto deixarei isso como um exercício para o leitor.

Esses exemplos mostram que a frase “Prefira composição de objetos à herança de classe” poderia estar mais bem formulada como “Prefira uma mistura criteriosa de composição e herança a usar somente uma delas” – mas acho que não é uma frase de efeito tão boa assim.

Substituir superclasse por delegação (Replace Superclass with Delegate)

anteriormente: *Substituir herança por delegação* (Replace Inheritance with Delegation)



```
class List {...}
class Stack extends List
{...}
```

```
class Stack {
  constructor() {
    this._storage = new
List();
  }
}
class List {...}
```

Motivação

Em programas orientados a objetos, a herança é uma forma simples e eficaz de reutilizar funcionalidades. Herdo de alguma classe existente, e então sobrescrevo e acrescento outros recursos. Porém, a criação de subclasses pode ser feita de um modo que resulte em confusão e em complicações.

Um dos exemplos clássicos de uso indevido de herança no início da era dos objetos era fazer com que uma pilha (stack) fosse uma subclasse de lista (list). A ideia que levava a isso era a reutilização da armazenagem de dados e das operações para manipular a lista. Embora fosse bom reutilizar código, essa herança apresentava um problema: todas as operações de lista estavam presentes na interface da pilha, embora a maioria delas não lhe fosse aplicável. Uma abordagem melhor é fazer com que a lista seja um campo da

pilha e delegar as operações necessárias a ela.

Esse é um exemplo de um dos motivos para usar *Substituir superclasse por delegação* – se as funções da superclasse não fizerem sentido na subclasse, é sinal de que eu não deveria estar usando herança para ter as funcionalidades da superclasse.

Além de usar todas as funções da superclasse, também deve ser verdade que toda instância da subclasse é uma instância da superclasse, e ela deve ser um objeto válido em todos os casos em que estivermos usando a superclasse. Se eu tiver uma classe para o modelo de um carro, com informações como nome e capacidade do motor, posso pensar que seria possível reutilizar esses recursos para representar um carro físico, adicionando funções para o Número de Identificação do Veículo (VIN number) e para a data de fabricação. Esse é um erro de modelagem comum, muitas vezes sutil, que chamo de homônimo entre tipo e instância (type-instance homonym) [mf-tih].

Ambos são exemplos de problemas que resultam em confusão e em erros – esses podem ser facilmente evitados substituindo a herança por delegação para um objeto distinto. Usar delegação deixa claro que o objeto é diferente – um objeto para o qual somente algumas das funções se aplicam.

Mesmo nos casos em que a subclasse seja uma opção razoável de modelagem, uso *Substituir superclasse por delegação* porque o relacionamento entre uma subclasse e uma superclasse é altamente acoplado, com a subclasse apresentando falhas facilmente como consequência de mudanças na superclasse. A desvantagem é a necessidade de escrever um método de encaminhamento para qualquer função que seja igual no host e na classe de delegação – felizmente, porém, apesar de ser tedioso escrever essas funções de encaminhamento, elas são muito simples, e será muito difícil cometer erros aí.

Como consequência disso tudo, algumas pessoas aconselham a evitar totalmente a herança – mas não concordo. Desde que as condições semânticas apropriadas se apliquem (todo método do supertipo se aplica ao subtipo, toda instância do subtipo é uma instância do supertipo), a herança é um mecanismo simples e eficaz. Posso facilmente aplicar *Substituir superclasse por delegação* caso a situação mude e a herança deixe de ser a melhor opção. Assim, meu conselho é usar herança (na maioria das vezes) antes, e aplicar *Substituir superclasse por delegação* quando (e se) ela se tornar um problema.

Procedimento

- Crie um campo na subclasse que faça referência ao objeto da superclasse. Inicialize essa referência à delegação com uma nova instância.
- Para cada elemento da superclasse, crie uma função na subclasse que faça um encaminhamento para a referência da delegação. Teste após fazer o encaminhamento para cada grupo consistente.

  Na maioria das vezes, você poderá testar depois de cada função encaminhada, mas pares get/set, por exemplo, só poderão ser testados depois que ambos tiverem sido movidos.
- Quando todos os elementos da superclasse tiverem sido sobrescritos pelas funções de encaminhamento, remova a ligação de herança.

Exemplo

Recentemente dei uma consultoria para uma biblioteca de pergaminhos antigos (ancient scrolls) de uma velha cidade. Ela mantém detalhes sobre seus pergaminhos em um catálogo. Cada pergaminho tem um número de ID e tem seu título e uma lista de tags registrados.

class CatalogItem…

```
constructor(id, title, tags) {
  this._id = id;
  this._title = title;
  this._tags = tags;
}

get id() {return this._id;}
get title() {return this._title;}
hasTag(arg) {return this._tags.includes(arg);}
```

Uma das exigências dos pergaminhos é uma limpeza frequente. O código para isso utiliza o item do catálogo e o estende com os dados necessários para a limpeza.

class Scroll extends CatalogItem…

```
constructor(id, title, tags, dateLastCleaned) {
  super(id, title, tags);
  this._lastCleaned = dateLastCleaned;
}

needsCleaning(targetDate) {
```

```
    const threshold = this.hasTag("revered") ? 700 : 1500;
    return this.daysSinceLastCleaning(targetDate) > threshold ;
  }
  daysSinceLastCleaning(targetDate) {
    return this._lastCleaned.until(targetDate, ChronoUnit.DAYS);
  }
```

Esse é um exemplo de um erro comum de modelagem. Há uma diferença entre o pergaminho físico e o item do catálogo. O pergaminho que descreve o tratamento para peste cinzenta (greyscale disease) pode ter várias cópias, mas pode ser um único item no catálogo.

Em várias situações, um erro como esse talvez seja tolerável. Posso pensar no título (title) e nas tags como cópias dos dados do catálogo. Caso esses dados não mudem nunca, essa representação talvez seja aceitável. No entanto, se eu tiver de atualizar algum deles, devo ser cuidadoso e garantir que todas as cópias do mesmo item do catálogo sejam atualizadas corretamente.

Mesmo sem esse problema, eu ainda gostaria de modificar o relacionamento. Usar um item do catálogo como uma superclasse para um pergaminho provavelmente confundirá os programadores no futuro e, desse modo, é um modelo ruim para trabalhar.

Começo criando uma propriedade em Scroll que faça referência ao item do catálogo, inicializando-o com uma nova instância.

class Scroll extends CatalogItem…

```
  constructor(id, title, tags, dateLastCleaned) {
    super(id, title, tags);
    this._catalogItem = new CatalogItem(id, title, tags);
    this._lastCleaned = dateLastCleaned;
  }
```

Crio métodos de encaminhamento para cada elemento da superclasse que uso na subclasse.

class Scroll…

```
  get id() {return this._catalogItem.id;}
  get title() {return this._catalogItem.title;}
  hasTag(aString) {return this._catalogItem.hasTag(aString);}
```

Removo a ligação de herança com o item do catálogo.

```
  class Scroll ~~extends CatalogItem~~{
    constructor(id, title, tags, dateLastCleaned) {
      ~~super(id, title, tags);~~
```

```
    this._catalogItem = new CatalogItem(id, title, tags);
    this._lastCleaned = dateLastCleaned;
  }
```

Romper a ligação de herança encerra a refatoração *Substituir superclasse por delegação* básica, mas há algo mais que devo fazer nesse caso.

A refatoração muda o papel do item do catálogo, que passa a ser um componente do pergaminho; cada pergaminho contém uma instância única de um item de catálogo. Em muitos casos em que faço essa refatoração, isso é suficiente. No entanto, nesse exemplo, um modelo melhor seria associar o item do catálogo para a peste cinzenta aos seis pergaminhos na biblioteca que são cópias daquele texto. Fazer isso é essencialmente aplicar *Mudar valor para referência (Change Value to Reference)*.

Há um problema que devo corrigir, porém, antes de usar *Mudar valor para referência (Change Value to Reference)*. Na estrutura original de herança, o pergaminho usava o campo de ID do item do catálogo para armazenar o seu ID. No entanto, se eu tratar o item do catálogo como uma referência, será necessário usar esse ID do item do catálogo em vez de usar o ID do pergaminho. Isso significa que devo criar um campo de ID no pergaminho e usá-lo no lugar do ID do item do catálogo. É uma operação meio parecida com mover, meio parecida com separar.

class Scroll…

```
constructor(id, title, tags, dateLastCleaned) {
  this._id = id;
  this._catalogItem = new CatalogItem(null, title, tags);
  this._lastCleaned = dateLastCleaned;
}
get id() {return this._id;}
```

Criar um item de catálogo com um ID nulo em geral dispararia alertas vermelhos e faria alarmes soarem. Porém, a solução é apenas temporária, enquanto estou trabalhando. Depois de fazer isso, os pergaminhos farão referência a um item de catálogo compartilhado, com o seu próprio ID.

Atualmente, os pergaminhos são carregados como parte de uma rotina de carga.

rotina de carga…

```
const scrolls = aDocument
      .map(record => new Scroll(record.id,
                                record.catalogData.title,
```

```
                record.catalogData.tags,
                LocalDate.parse(record.lastCleaned)));
```

O primeiro passo em *Mudar valor para referência (Change Value to Reference)* é encontrar ou criar um repositório. Noto que há um repositório que posso facilmente importar na rotina de carga. O repositório disponibiliza itens de catálogo indexados por um ID. Minha próxima tarefa é verificar como posso disponibilizar esse ID no construtor do pergaminho. Felizmente, ele está presente nos dados de entrada, embora fosse ignorado, pois não tinha utilidade quando a herança era usada. Com essa questão resolvida, posso agora usar *Mudar declaração de função (Change Function Declaration)* para adicionar tanto o catálogo quanto o ID do item do catálogo nos parâmetros do construtor.

rotina de carga…

```
const scrolls = aDocument
    .map(record => new Scroll(record.id,
                record.catalogData.title,
                record.catalogData.tags,
                LocalDate.parse(record.lastCleaned),
                record.catalogData.id,
                catalog));
```

class Scroll…

```
constructor(id, title, tags, dateLastCleaned, catalogID, catalog) {
  this._id = id;
  this._catalogItem = new CatalogItem(null, title, tags);
  this._lastCleaned = dateLastCleaned;
}
```

Agora modifico o construtor para que use o ID do catálogo a fim de procurar o item de catálogo e utilizá-lo, em vez de criar um novo ID.

class Scroll…

```
constructor(id, title, tags, dateLastCleaned, catalogID, catalog) {
  this._id = id;
  this._catalogItem = catalog.get(catalogID);
  this._lastCleaned = dateLastCleaned;
}
```

Não é mais necessário que o título e as tags sejam passados para o construtor, portanto uso *Mudar declaração de função (Change Function Declaration)* para removê-los.

load routine…

```
const scrolls = aDocument
      .map(record => new Scroll(record.id,
                                record.catalogData.title,
                                record.catalogData.tags,
                                LocalDate.parse(record.lastCleaned),
                                record.catalogData.id,
                                catalog));
```

class Scroll…

```
  constructor(id, title, tags, dateLastCleaned, catalogID, catalog) {
    this._id = id;
    this._catalogItem = catalog.get(catalogID);
    this._lastCleaned = dateLastCleaned;
  }
```

Bibliografia

Você pode encontrar uma versão online desta bibliografia em https://martinfowler.com/books/refactoring-bibliography.html. Muitas das entradas aqui apresentadas se referem à "bliki" – uma seção de martinfowler.com em que apresento descrições concisas de vários termos usados em desenvolvimento de software. Quando escrevi este livro, decidi encaminhar os leitores às explicações que eu havia disponibilizado ali, em vez de incorporá-las no texto do livro.

[Ambler & Sadalage] Scott W. Ambler e Pramod J. Sadalage. *Refactoring Databases*. Addison-Wesley, 2006. ISBN 0321293533.

[babel] https://babeljs.io.

[Bazuzi] Jay Bazuzi. "Safely Extract a Method in Any C++ Code" (Como extrair um método de forma segura em qualquer código C++).

http://jay.bazuzi.com/Safely-extract-a-method-in-any-C++-code/.

[Beck SBPP] Kent Beck. *Smalltalk Best Practice Patterns*. Addison-Wesley, 1997. ISBN 013476904X.

[chai] http://chaijs.com.

[eclipse] http://www.eclipse.org.

[Feathers] Michael Feathers. *Trabalho eficaz com código legado*. Bookman, 2013.

[Fields et al.] Jay Fields, Shane Harvie e Martin Fowler. *Refactoring Ruby Edition*. Addison-Wesley, 2009. ISBN 0321603508.

[Ford et al.] Neal Ford, Rebecca Parsons e Patrick Kua. *Building Evolutionary Architectures*. O'Reilly, 2017. ISBN 1491986360.

[Forsgren et al.] Nicole Forsgren, Jez Humble e Gene Kim. *Accelerate: The Science of Lean Software and DevOps: Building and Scaling High Performing Technology Organizations*. IT Revolution Press, 2018. ISBN 1942788339.

[gof] Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides. *Padrões de projeto: soluções reutilizáveis de software orientados a objetos*.

Bookman, 2000.

[Harold] Elliotte Rusty Harold. *Refatorando HTML*. Bookman, 2010.

[intellij] https://www.jetbrains.com/idea/.

[Kerievsky] Joshua Kerievsky. *Refatoração para padrões*. Bookman, 2008.

[langserver] https://langserver.org.

[maudite] https://en.wikipedia.org/wiki/Unibroue.

[mf-2h] Martin Fowler. “Bliki: TwoHardThings” (Duas tarefas difíceis). https://martinfowler.com/bliki/TwoHardThings.html.

[mf-bba] Martin Fowler. “Bliki: BranchByAbstraction” (Branch por abstração). https://martinfowler.com/bliki/BranchByAbstraction.html.

[mf-cp] Martin Fowler. “Collection Pipeline” (Pipeline de coleção). https://martinfowler.com/articles/collection-pipeline/.

[mf-cqs] Martin Fowler. “Bliki: CommandQuerySeparation” (Separação entre comandos-consultas). https://martinfowler.com/bliki/CommandQuerySeparation.html.

[mf-cw] Martin Fowler. “Bliki: ClockWrapper” (Encapsulador de relógio). https://martinfowler.com/bliki/ClockWrapper.html.

[mf-dsh] Martin Fowler. “Bliki: DesignStaminaHypothesis” (Hipótese da Estamina no Design). https://martinfowler.com/bliki/DesignStaminaHypothesis.html.

[mf-evodb] Pramod Sadalage e Martin Fowler. “Evolutionary Database Design” (Design de banco de dados evolucionário). https://martinfowler.com/articles/evodb.html.

[mf-fao] Martin Fowler. “Bliki: FunctionAsObject” (Função como objeto). https://martinfowler.com/bliki/FunctionAsObject.html.

[mf-ft] Martin Fowler. “Form Template Method” (Formar método de template). https://refactoring.com/catalog/formTemplateMethod.html.

[mf-lh] Martin Fowler. “Bliki: ListAndHash” (Lista-e-hash). https://martinfowler.com/bliki/ListAndHash.html.

[mf-nm] Martin Fowler. “The New Methodology” (A nova metodologia). https://martinfowler.com/articles/newMethodology.html.

[mf-ogs] Martin Fowler. “Bliki: OverloadedGetterSetter” (Getter e setter sobrecarregados).

https://martinfowler.com/bliki/OverloadedGetterSetter.html.

[mf-pc] Danilo Sato. “Bliki: ParallelChange” (Mudança em paralelo).

https://martinfowler.com/bliki/ParallelChange.html.

[mf-range] Martin Fowler. “Range” (Intervalo).

https://martinfowler.com/eaaDev/Range.html.

[mf-ref-doc] Martin Fowler. “Refactoring Code to Load a Document” (Refatorando código para carregar um documento).

https://martinfowler.com/articles/refactoring-document-load.html.

[mf-ref-pipe] Martin Fowler. “Refactoring with Loops and Collection Pipelines” (Refatoração com laços e pipelines de coleção).

https://martinfowler.com/articles/refactoring-pipelines.html.

[mf-repos] Martin Fowler. “Repository” (Repositório).

https://martinfowler.com/eaaCatalog/repository.html.

[mf-stc] Martin Fowler. “Bliki: SelfTestingCode” (Código autotestável).

https://martinfowler.com/bliki/SelfTestingCode.html.

[mf-tc] Martin Fowler. “Bliki: TestCoverage” (Cobertura de testes).

https://martinfowler.com/bliki/TestCoverage.html.

[mf-tdd] Martin Fowler. “Bliki: TestDrivenDevelopment” (Desenvolvimento orientado a testes).

https://martinfowler.com/bliki/TestDrivenDevelopment.html.

[mf-tih] Martin Fowler. “Bliki: TypeInstanceHomonym” (Homônimo entre tipo e instância).

https://martinfowler.com/bliki/TypeInstanceHomonym.html.

[mf-ua] Martin Fowler. “Bliki: UniformAccessPrinciple” (Princípio do Acesso Uniforme).

https://martinfowler.com/bliki/UniformAccessPrinciple.html.

[mf-vo] Martin Fowler. “Bliki: ValueObject” (Objeto de valor). https://martinfowler.com/bliki/ValueObject.html.

[mf-xp] Martin Fowler. “Bliki: ExtremeProgramming”.

https://martinfowler.com/bliki/ExtremeProgramming.html.

[mf-xunit] Martin Fowler. “Bliki: Xunit”. https://martinfowler.com/bliki/Xunit.html.

[mf-yagni] Martin Fowler. “Bliki: Yagni”. https://martinfowler.com/bliki/Yagni.html.

[mocha] https://mochajs.org.

[Opdyke] William F. Opdyke. “Refactoring Object-Oriented Frameworks” (Refatoração de frameworks orientados a objetos). Dissertação de doutorado. Universidade de Illinois em Urbana-Champaign, 1992.

http://www.laputan.org/pub/papers/opdyke-thesis.pdf.

[Parnas] D. L. Parnas. “On the Criteria to Be Used in Decomposing Systems into Modules” (Sobre os critérios a serem usados na decomposição de sistemas em módulos). In: *Communications of the ACM, v. 15, n. 12, p. 1053–1058*. Dez. 1972.

[ref.com] https://refactoring.com.

[Wake] William C. Wake. *Refactoring Workbook*. Addison-Wesley, 2003. ISBN 0321109295.

[wake-swap] Bill Wake. “The Swap Statement Refactoring” (A refatoração *Trocar instruções*). https://www.industriallogic.com/blog/swap-statement-refactoring/.

Lista de refatorações

Combinar funções em classe (Combine Functions into Class)
Combinar funções em transformação (Combine Functions into Transform)
Condensar Hierarquia (Collapse Hierarchy)
Consolidar expressão condicional (Consolidate Conditional Expression)
Decompor condicional (Decompose Conditional)
Descer campo (Push Down Field)
Descer método (Push Down Method)
Deslocar instruções (Slide Statements)
Dividir laço (Split Loop)
Encapsular coleção (Encapsulate Collection)
Encapsular registro (Encapsulate Record)
Encapsular variável (Encapsulate Variable)
Extrair classe (Extract Class)
Extrair função (Extract Function)
Extrair superclasse (Extract Superclass)
Extrair variável (Extract Variable)
Agrupamentos de dados
Internalizar função (Inline Function)
Internalizar variável (Inline Variable)
Introduzir asserção (Introduce Assertion)
Introduzir caso especial (Introduce Special Case)
Introduzir objeto de parâmetros (Introduce Parameter Object)
Mover campo (Move Field)
Mover função (Move Function)
Mover instruções para os pontos de chamada (Move Statements to Callers)
Mover instruções para uma função (Move Statements into Function)

Mudar declaração de função (Change Function Declaration)
Mudar referência para valor (Change Reference to Value)
Mudar valor para referência (Change Value to Reference)
Ocultar delegação (Hide Delegate)
Parametrizar função (Parameterize Function)
Preservar objeto inteiro (Preserve Whole Object)
Remover argumento de flag (Remove Flag Argument)
Renomear campo (Rename Field)
Remover código morto (Remove Dead Code)
Remover intermediário (Remove Middle Man)
Remover método de escrita (Remove Setting Method)
Remover subclasse (Remove Subclass)
Renomear variável (Rename Variable)
Separar consulta de modificador (Separate Query from Modifier)
Separar em fases (Split Phase)
Separar variável (Split Variable)
Subir campo (Pull Up Field)
Subir corpo do construtor (Pull Up Constructor Body)
Subir método (Pull Up Method)
Substituir algoritmo (Substitute Algorithm)
Substituir comando por função (Replace Command with Function)
Substituir condicional aninhada por cláusulas de guarda (Replace Nested Conditional with Guard Clauses)
Substituir condicional por polimorfismo (Replace Conditional with Polymorphism)
Substituir construtor por função de factory (Replace Constructor with Factory Function)
Substituir consulta por parâmetro (Replace Query with Parameter)
Substituir código de tipos por subclasses (Replace Type Code with Subclasses)

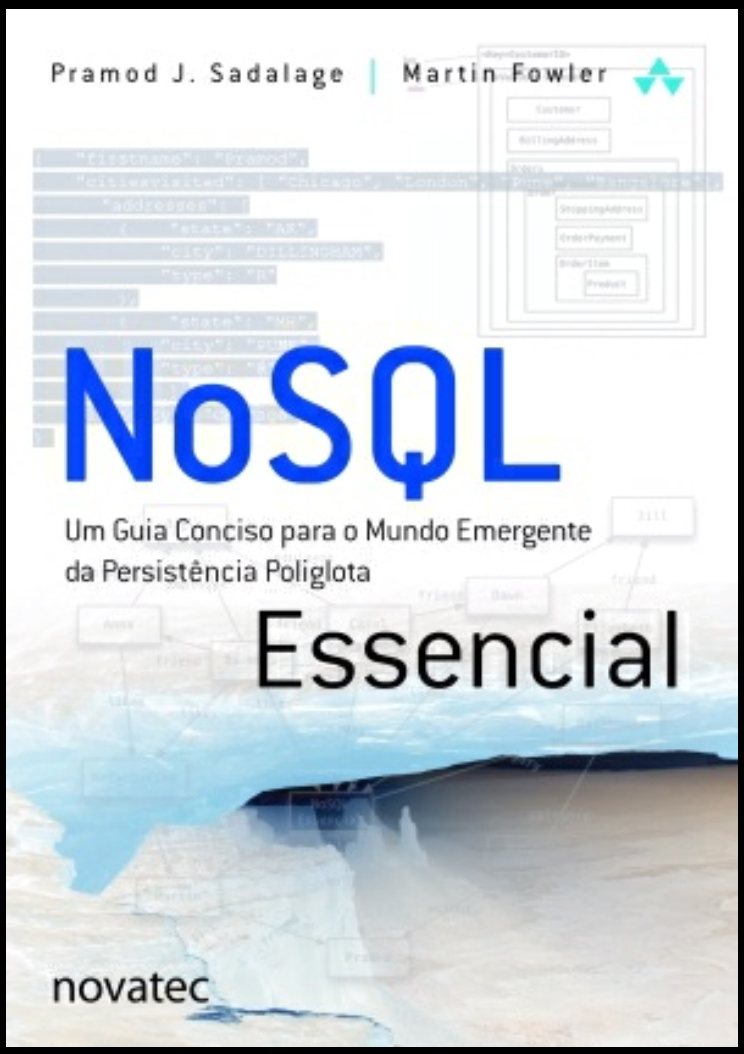
Substituir código internalizado por chamada de função (Replace Inline Code with Function Call)
Substituir função por comando (Replace Function with Command)
Substituir laço por pipeline (Replace Loop with Pipeline)
Substituir parâmetro por consulta (Replace Parameter with Query)
Substituir primitivo por objeto (Replace Primitive with Object)
Substituir subclasse por delegação (Replace Subclass with Delegate)
Substituir superclasse por delegação (Replace Superclass with Delegate)
Substituir variável derivada por consulta (Replace Derived Variable with Query)
Substituir variável temporária por consulta (Replace Temp with Query)

“Maus cheiros” no código

“Mau cheiro” (smell)	Refatorações comuns
Agrupamentos de dados	*Extrair classe (Extract Class)*, *Introduzir objeto de parâmetros (Introduce Parameter Object)*, *Preservar objeto inteiro (Preserve Whole Object)*
Alteração divergente	*Separar em fases (Split Phase)*, *Mover função (Move Function)*, *Extrair função (Extract Function)*, *Extrair classe (Extract Class)*
Cadeias de mensagens	*Ocultar delegação (Hide Delegate)*, *Extrair função (Extract Function)*, *Mover função (Move Function)*
Campo temporário	*Extrair classe (Extract Class)*, *Mover função (Move Function)*, *Introduzir caso especial (Introduce Special Case)*
Cirurgia com rifle	*Mover função (Move Function)*, *Mover campo (Move Field)*, *Combinar funções em classe (Combine Functions into Class)*, *Combinar funções em transformação (Combine Functions into Transform)*, *Separar em fases (Split Phase)*, *Internalizar função (Inline Function)*, *Internalizar classe (Inline Class)*
Classe de dados	*Encapsular registro (Encapsulate Record)*, *Remover método de escrita (Remove Setting Method)*, *Mover função (Move Function)*, *Extrair função (Extract Function)*, *Separar em fases (Split Phase)*
Classe grande	*Extrair classe (Extract Class)*, *Extrair superclasse (Extract Superclass)*, *Substituir código de tipos por subclasses (Replace Type Code with Subclasses)*
Classes alternativas com interfaces diferentes	*Mudar declaração de função (Change Function Declaration)*, *Mover função (Move Function)*, *Extrair superclasse (Extract Superclass)*
Comentários	*Extrair função (Extract Function)*, *Mudar declaração de*

	função (Change Function Declaration), *Introduzir asserção (Introduce Assertion)*
Código duplicado	*Extrair função (Extract Function)*, *Deslocar instruções (Slide Statements)*, *Subir método (Pull Up Method)*
Dados globais	*Encapsular variável (Encapsulate Variable)*
Dados mutáveis	*Encapsular variável (Encapsulate Variable)*, *Separar variável (Split Variable)*, *Deslocar instruções (Slide Statements)*, *Extrair função (Extract Function)*, *Separar consulta de modificador (Separate Query from Modifier)*, *Remover método de escrita (Remove Setting Method)*, *Substituir variável derivada por consulta (Replace Derived Variable with Query)*, *Combinar funções em classe (Combine Functions into Class)*, *Combinar funções em transformação (Combine Functions into Transform)*, *Mudar referência para valor (Change Reference to Value)*
Elemento ocioso	*Internalizar função (Inline Function)*, *Internalizar classe (Inline Class)*, *Condensar Hierarquia (Collapse Hierarchy)*
Função longa	*Extrair função (Extract Function)*, *Substituir variável temporária por consulta (Replace Temp with Query)*, *Introduzir objeto de parâmetros (Introduce Parameter Object)*, *Preservar objeto inteiro (Preserve Whole Object)*, *Substituir função por comando (Replace Function with Command)*, *Decompor condicional (Decompose Conditional)*, *Substituir condicional por polimorfismo (Replace Conditional with Polymorphism)*, *Dividir laço (Split Loop)*
Generalidade especulativa	*Condensar Hierarquia (Collapse Hierarchy)*, *Internalizar função (Inline Function)*, *Internalizar classe (Inline Class)*, *Mudar declaração de função (Change Function Declaration)*, *Remover código morto (Remove Dead Code)*
Herança recusada	*Descer método (Push Down Method)*, *Descer campo (Push Down Field)*, *Substituir subclasse por delegação (Replace Subclass with Delegate)*, *Substituir superclasse por delegação (Replace Superclass with Delegate)*

Intermediário	*Remover intermediário (Remove Middle Man)*, *Internalizar função (Inline Function)*, *Substituir superclasse por delegação (Replace Superclass with Delegate)*, *Substituir subclasse por delegação (Replace Subclass with Delegate)*
Inveja de recursos	*Mover função (Move Function)*, *Extrair função (Extract Function)*
Laços	*Substituir laço por pipeline (Replace Loop with Pipeline)*
Lista longa de parâmetros	*Substituir parâmetro por consulta (Replace Parameter with Query)*, *Preservar objeto inteiro (Preserve Whole Object)*, *Introduzir objeto de parâmetros (Introduce Parameter Object)*, *Remover argumento de flag (Remove Flag Argument)*, *Combinar funções em classe (Combine Functions into Class)*
Nome misterioso	*Mudar declaração de função (Change Function Declaration)*, *Renomear variável (Rename Variable)*, *Renomear campo (Rename Field)*
Obsessão por primitivos	*Substituir primitivo por objeto (Replace Primitive with Object)*, *Substituir código de tipos por subclasses (Replace Type Code with Subclasses)*, *Substituir condicional por polimorfismo (Replace Conditional with Polymorphism)*, *Extrair classe (Extract Class)*, *Introduzir objeto de parâmetros (Introduce Parameter Object)*
Switches repetidos	*Substituir condicional por polimorfismo (Replace Conditional with Polymorphism)*
Trocas escusas	*Mover função (Move Function)*, *Mover campo (Move Field)*, *Ocultar delegação (Hide Delegate)*, *Substituir subclasse por delegação (Replace Subclass with Delegate)*, *Substituir superclasse por delegação (Replace Superclass with Delegate)*



NoSQL Essencial

Sadalage, Pramod J.

9788575227855

216 páginas

A necessidade de se lidar com volumes cada vez maiores de

dados é um fator que motiva adotar uma nova classe de bancos de dados não relacionais, NoSQL. Os defensores dos bancos de dados NoSQL alegam que estes podem ser utilizados para criar sistemas com melhor desempenho, escalabilidade e mais fáceis de programar. NoSQL Essencial é uma introdução concisa, porém completa, a essa tecnologia emergente em rápida ascensão. Pramod J. Sadalage e Martin Fowler explicam como bancos de dados NoSQL funcionam e as formas pelas quais podem ser uma alternativa superior a um sistema tradicional de gerenciamento de banco de dados relacional. Os autores apresentam um guia rápido sobre os conceitos que você precisa conhecer para avaliar se os bancos de dados NoSQL são apropriados a suas necessidades e, se forem, quais tecnologias você deve explorar mais detalhadamente. A primeira parte do livro concentra-se em conceitos básicos, incluindo modelos de dados sem esquema, agregados, novos modelos de distribuição, teorema CAP e map-reduce. Na segunda parte, os autores exploram questões de arquitetura e projeto associadas à implementação de NoSQL. Também apresentam casos de uso reais, que mostram os bancos de dados NoSQL em ação, e fornecem exemplos representativos utilizando Riak, MongoDB, Cassandra e Neo4j. Além disso, ao trazer o trabalho pioneiro de Pramod Sadalage, NoSQL Essencial mostra como

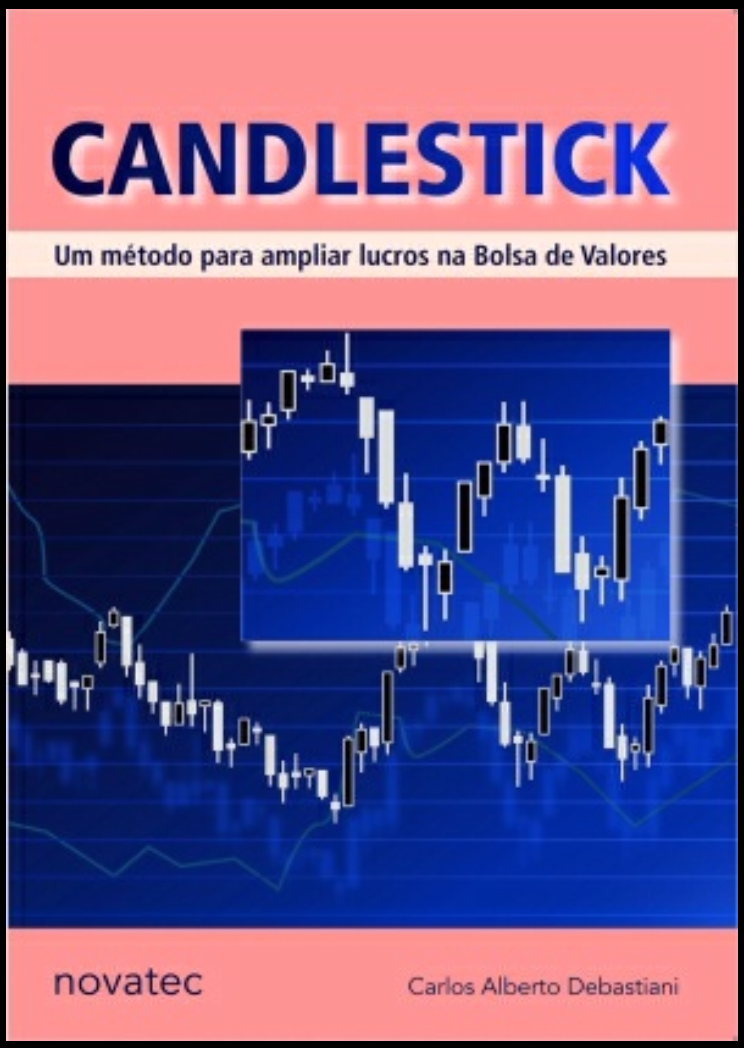
implementar projetos de forma evolutiva com migração de esquemas: uma técnica fundamental a ser utilizada em bancos de dados NoSQL. O livro termina descrevendo como o NoSQL se estabelece em uma nova era de persistência poliglota, em que diversos mundos de armazenamento de dados coexistem e os arquitetos podem escolher a tecnologia mais otimizada para cada tipo de acesso a dados.

Programação web com Node e Express

Brown, Ethan
9786586057096
368 páginas

Construa aplicações web dinâmicas com o Express, um

componente-chave da stack de desenvolvimento Node/JavaScript. Nesta edição atualizada, o autor Ethan Brown ensina os fundamentos do Express 5 percorrendo o desenvolvimento de uma aplicação. Este guia prático aborda de tudo, da renderização no lado do servidor ao desenvolvimento de uma API adequada para uso em aplicativos de página única (SPAs). O Express conseguiu chegar a um equilíbrio entre um framework robusto e a ausência de framework, permitindo-nos fazer escolhas livres para nossa arquitetura. Engenheiros front-end e back-end familiarizados com JavaScript também aprenderão práticas recomendadas para a construção de aplicações web de várias páginas e híbridos com o Express. Compre este livro e descubra novas maneiras de olhar para o desenvolvimento web.



Candlestick

Debastiani, Carlos Alberto

9788575225943
200 páginas

A análise dos gráficos de Candlestick é uma técnica

amplamente utilizada pelos operadores de bolsas de valores no mundo inteiro. De origem japonesa, este refinado método avalia o comportamento do mercado, sendo muito eficaz na previsão de mudanças em tendências, o que permite desvendar fatores psicológicos por trás dos gráficos, incrementando a lucratividade dos investimentos. Candlestick – Um método para ampliar lucros na Bolsa de Valores é uma obra bem estruturada e totalmente ilustrada. A preocupação do autor em utilizar uma linguagem clara e acessível a torna leve e de fácil assimilação, mesmo para leigos. Cada padrão de análise abordado possui um modelo com sua figura clássica, facilitando a identificação. Depois das características, das peculiaridades e dos fatores psicológicos do padrão, é apresentado o gráfico de um caso real aplicado a uma ação negociada na Bovespa. Este livro possui, ainda, um índice resumido dos padrões para pesquisa rápida na utilização cotidiana.

Avaliando Empresas, Investindo em Ações

Debastiani, Carlos Alberto
9788575225974
224 páginas

Avaliando Empresas, Investindo em Ações é um livro destinado a investidores que desejam conhecer, em detalhes, os métodos de análise que integram a linha de trabalho da escola fundamentalista, trazendo ao leitor, em linguagem clara e acessível, o conhecimento profundo dos elementos necessários a uma análise criteriosa da saúde financeira das empresas, envolvendo indicadores de balanço e de mercado, análise de liquidez e dos riscos pertinentes a fatores setoriais e conjunturas econômicas nacional e internacional. Por meio de exemplos práticos e ilustrações, os autores exercitam os conceitos teóricos abordados, desde os fundamentos básicos da economia até a formulação de estratégias para investimentos de longo prazo.

Manual de Análise Técnica

Abe, Marcos
9788575227022
256 páginas

Este livro aborda o tema Investimento em Ações de maneira

inédita e tem o objetivo de ensinar os investidores a lucrarem nas mais diversas condições do mercado, inclusive em tempos de crise. Ensinará ao leitor que, para ganhar dinheiro, não importa se o mercado está em alta ou em baixa, mas sim saber como operar em cada situação. Com o Manual de Análise Técnica o leitor aprenderá: - os conceitos clássicos da Análise Técnica de forma diferenciada, de maneira que assimile não só os princípios, mas que desenvolva o raciocínio necessário para utilizar os gráficos como meio de interpretar os movimentos da massa de investidores do mercado; - identificar oportunidades para lucrar na bolsa de valores, a longo e curto prazo, até mesmo em mercados baixistas; um sistema de investimentos completo com estratégias para abrir, conduzir e fechar operações, de forma que seja possível maximizar lucros e minimizar prejuízos; - estruturar e proteger operações por meio do gerenciamento de capital. Destina-se a iniciantes na bolsa de valores e investidores que ainda não desenvolveram uma metodologia própria para operar lucrativamente.