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

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;
```


image
not
available

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) {
```


image
not
available

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}`);
  }
```

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);
```

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

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

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 –

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.

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