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: 978-85-7522-724-4

Histórico de impressões:

Abril/2020 Primeira edição

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
FC20200420

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) (134)*.

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) (134) 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) (134)*, 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) (207)*.

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) (152)*.

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) (153)* 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) (152)*.

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) (153)*.

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) (257)* 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) (252)* 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) (207)*. Como antes, o primeiro passo é aplicar *Extrair função (Extract Function) (134)* 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) (152)*:

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) (257)* para isolar a acumulação;
- *Deslocar instruções (Slide Statements) (252)* para levar o código de inicialização para perto da acumulação;
- *Extrair função (Extract Function) (134)* para criar uma função que calcula o total;
- *Internalizar variável (Inline Variable) (152)* 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) (183)*. 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) (183)* aplicando *Extrair função (Extract Function) (134)* 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) (225)* em `playFor` e em `statement` (compilo-testo-faço commit).

function statement…

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

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

function renderPlainText…

```
  let result = `Statement for ${data.customer}\n`;
  for (let perf of data.performances) {
    result += `  ${perf.play.name}: ${usd(amountFor(perf))} (${perf.audience} seats)\n`;
  }
  result += `Amount owed is ${usd(totalAmount())}\n`;
  result += `You earned ${totalVolumeCredits()} credits\n`;
  return result;
function volumeCreditsFor(aPerformance) {
  let result = 0;
  result += Math.max(aPerformance.audience - 30, 0);
  if ("comedy" === aPerformance.play.type) result += Math.floor(aPerformance.audience / 5);
  return result;
}
```

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

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

function statement…

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

function renderPlainText…

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

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

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

function statement…

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

function renderPlainText…

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

Por fim, movo os dois cálculos de totais.

function statement…

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

function renderPlainText…

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

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

Depois de compilar-testar-fazer commit após a mudança, não consegui resistir a alguns usos rápidos de *Substituir laço por pipeline (Replace Loop with Pipeline) (261)*.

function renderPlainText…

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

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

nível mais alto…

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

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

statement.js…

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

createStatementData.js…

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

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

statement.js…

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

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

Status: separado em dois arquivos (e fases)

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

statement.js

```
import createStatementData from './createStatementData.js';
function statement (invoice, plays) {
  return renderPlainText(createStatementData(invoice, plays));
}
function renderPlainText(data, plays) {
  let result = `Statement for ${data.customer}\n`;
  for (let perf of data.performances) {
    result += `  ${perf.play.name}: ${usd(perf.amount)} (${perf.audience} seats)\n`;
  }
  result += `Amount owed is ${usd(data.totalAmount)}\n`;
  result += `You earned ${data.totalVolumeCredits} credits\n`;
  return result;
}
function htmlStatement (invoice, plays) {
  return renderHtml(createStatementData(invoice, plays));
}
function renderHtml (data) {
  let result = `<h1>Statement for ${data.customer}</h1>\n`;
  result += "<table>\n";
  result += "<tr><th>play</th><th>seats</th><th>cost</th></tr>";
  for (let perf of data.performances) {
    result += `  <tr><td>${perf.play.name}</td><td>${perf.audience}</td>`;
    result += `<td>${usd(perf.amount)}</td></tr>\n`;
  }
  result += "</table>\n";
  result += `<p>Amount owed is <em>${usd(data.totalAmount)}</em></p>\n`;
  result += `<p>You earned <em>${data.totalVolumeCredits}</em> credits</p>\n`;
  return result;
}
function usd(aNumber) {
  return new Intl.NumberFormat("en-US",
                               { style: "currency", currency: "USD",
                                 minimumFractionDigits: 2 }).format(aNumber/100);
}
```

createStatementData.js

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

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

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

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

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

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

Reorganizando os cálculos por tipo

Voltarei minha atenção agora para a próxima mudança de funcionalidade: aceitar outras categorias de peças, cada uma com seus próprios cálculos de cobrança e de créditos por volume. No momento, para fazer alterações, tenho de entrar nas funções de cálculo e editar ali as condições. A função `amountFor` enfatiza o papel central

que o tipo da peça desempenha na forma de fazer os cálculos – porém uma lógica condicional como essa tende a se enfraquecer à medida que novas modificações forem feitas, a menos que ela seja reforçada por elementos mais estruturais da linguagem de programação.

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

De modo geral, meu plano é definir uma hierarquia de herança com subclasses para comédia e tragédia que contenham a lógica de cálculo para esses casos. Quem fizer a chamada usará uma função polimórfica para o cálculo do valor, e a linguagem despachará para os diferentes cálculos associados às comédias e às tragédias. Criarei uma estrutura semelhante para o cálculo dos créditos por volume. Para isso, utilizarei duas refatorações. A refatoração principal é *Substituir condicional por polimorfismo (Replace Conditional with Polymorphism) (299)*, que altera uma porção de código com condicionais por polimorfismo. Contudo, antes de usar *Substituir condicional por polimorfismo*, é necessário criar algum tipo de estrutura de herança. Tenho de criar uma classe que contenha as funções para calcular o valor cobrado e os créditos por volume.

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

createStatementData.js…

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

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

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

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

Criando uma calculadora de apresentação

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

function createStatementData…

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

nível mais alto…

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

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

Para que isso funcione, usarei *Mudar declaração de função (Change Function Declaration) (153)* a fim de passar a peça encenada para a calculadora.

function createStatementData…

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

class PerformanceCalculator…

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

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

Passando funções para a calculadora

A próxima porção de lógica que moverei é muito maior e serve para calcular o valor de uma apresentação. Movi funções por aí casualmente enquanto reorganizava as funções aninhadas – mas esta é uma modificação mais profunda no contexto da função, portanto descreverei a refatoração *Mover função (Move Function) (225)* passo a passo. A primeira parte dessa refatoração é copiar a lógica para o seu novo contexto – a classe de calculadora. Em seguida, adapto o código para se adequar à sua nova morada, alterando `aPerformance` para `this.performance` e `playFor(aPerformance)` para `this.play`.

class PerformanceCalculator…

```
get amount() {
  let result = 0;
  switch (this.play.type) {
    case "tragedy":
      result = 40000;
      if (this.performance.audience > 30) {
        result += 1000 * (this.performance.audience - 30);
      }
      break;
    case "comedy":
      result = 30000;
      if (this.performance.audience > 20) {
        result += 10000 + 500 * (this.performance.audience - 20);
      }
      result += 300 * this.performance.audience;
      break;
```

```
      default:
        throw new Error(`unknown type: ${this.play.type}`);
    }
    return result;
  }
```

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

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

function createStatementData…

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

Agora posso compilar-testar-fazer commit para garantir que o código esteja funcionando de forma apropriada em sua nova morada. Depois de fazer isso, utilizo *Internalizar função (Inline Function) (144)* para chamar diretamente a nova função (compilo-testo-faço commit).

function createStatementData…

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

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

function createStatementData…

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

```
  result.volumeCredits = calculator.volumeCredits;
  return result;
}
```

class PerformanceCalculator…

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

Deixando a calculadora de apresentação polimórfica

Agora que tenho a lógica em uma classe, é hora de aplicar o polimorfismo. O primeiro passo é usar *Substituir código de tipos por subclasses (Replace Type Code with Subclasses) (389)* 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) (363)*.

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) (299)*.

Começo pelo cálculo do valor para as tragédias.

class TragedyCalculator…

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

Somente o fato de ter esse método na subclasse é suficiente para sobrescrever a condicional da superclasse. Todavia, se você for tão paranoico quanto eu, poderá fazer o seguinte:

class PerformanceCalculator…

```
get amount() {
  let result = 0;
  switch (this.play.type) {
    case "tragedy":
      throw 'bad thing';
    case "comedy":
      result = 30000;
      if (this.performance.audience > 20) {
```

```
        result += 10000 + 500 * (this.performance.audience - 20);
      }
      result += 300 * this.performance.audience;
      break;
    default:
      throw new Error(`unknown type: ${this.play.type}`);
  }
  return result;
}
```

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

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

class ComedyCalculator…

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

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

class PerformanceCalculator…

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

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

class PerformanceCalculator…

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

class ComedyCalculator…

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

Status: criando os dados com a calculadora polimórfica

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

createStatementData.js

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

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

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

function createPerformanceCalculator(aPerformance, aPlay) {
    switch(aPlay.type) {
    case "tragedy": return new TragedyCalculator(aPerformance, aPlay);
    case "comedy" : return new ComedyCalculator(aPerformance, aPlay);
    default:
        throw new Error(`unknown type: ${aPlay.type}`);
    }
}
class PerformanceCalculator {
  constructor(aPerformance, aPlay) {
    this.performance = aPerformance;
    this.play = aPlay;
  }
  get amount() {
    throw new Error('subclass responsibility');
  }
  get volumeCredits() {
    return Math.max(this.performance.audience - 30, 0);
  }
}
class TragedyCalculator extends PerformanceCalculator {
  get amount() {
    let result = 40000;
    if (this.performance.audience > 30) {
      result += 1000 * (this.performance.audience - 30);
    }
    return result;
  }
}
class ComedyCalculator extends PerformanceCalculator {
  get amount() {
    let result = 30000;
    if (this.performance.audience > 20) {
      result += 10000 + 500 * (this.performance.audience - 20);
    }
```

```
    result += 300 * this.performance.audience;
    return result;
  }
  get volumeCredits() {
    return super.volumeCredits + Math.floor(this.performance.audience / 5);
  }
}
```

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

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

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

Considerações finais

Este é um exemplo simples, mas espero que ele dê a você uma noção de como é uma refatoração. Usei várias refatorações, incluindo *Extrair função (Extract Function) (134)*, *Internalizar variável (Inline Variable) (152)*, *Mover função (Move Function) (225)* e *Substituir condicional por polimorfismo (Replace Conditional with Polymorphism) (299)*.

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) (183)* para separar o código de cálculos do código de exibição e, por fim, introdução de uma calculadora polimórfica para a lógica de cálculos. Cada uma delas acrescentou estrutura ao código, permitindo que eu comunicasse melhor o que o código estava fazendo.

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

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

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

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