# Refactoring: A First Example

*previous:*
Preface

*next:*
Principles in Refactoring

How do I begin to talk about refactoring? The traditional way is by introducing the history of the subject, broad principles, and the like. When somebody does that at a conference, get slightly sleepy. My mind starts wandering, with a low-priority background process polling the speaker until they give an example.

The examples wake me up because I can see what is going on. With principles, it is too easy to make broad generalizations—and too hard to figure out how to apply things. An example helps make things clear.

So I'm going to start this book with an example of refactoring. I'll talk about how refactoring works and will give you a sense of the refactoring process. I can then do the usual principles-style introduction in the next chapter.

With any introductory example, however, I run into a problem. If I pick a large program, describing it and how it is refactored is too complicated for a mortal reader to work through. (I tried this with the original book—and ended up throwing away two examples, which were still pretty small but took over a hundred pages each to describe.) However, i pick a program that is small enough to be comprehensible, refactoring does not look like is worthwhile.

I'm thus in the classic bind of anyone who wants to describe techniques that are useful fc real-world programs. Frankly, it is not worth the effort to do all the refactoring that I'm going to show you on the small program I will be using. But if the code I'm showing you is part of a larger system, then the refactoring becomes important. Just look at my example and imagine it in the context of a much larger system.

## The Starting Point

In the first edition of this book, my starting program printed a bill from a video rental store which may now lead many of you to ask: "What's a video rental store?" Rather than answer that question, I've reskinned the example to something that is both older and still current.

Image a company of theatrical players who go out to various events performing plays. Typically, a customer will request a few plays and the company charges them based on the size of the audience and the kind of play they perform. There are currently two kinds plays that the company performs: tragedies and comedies. As well as providing a bill for the performance, the company gives its customers "volume credits" which they can use for discounts on future performances—think of it as a customer loyalty mechanism.

The performers store data about their plays in a simple JSON file that looks something lik
this:

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

The data for their bills also comes in a JSON file:

*invoices.json…*
```
[
  {
    "customer": "BigCo",
    "performances": [
      {
        "playID": "hamlet",
        "audience": 55
      },
      {
        "playID": "as-like",
        "audience": 35
      },
      {
        "playID": "othello",
        "audience": 40
      }
    ]
  }
]
```

The code that prints the bill is this simple function:

```
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}`);
```

```
    }

    // add volume credits
    volumeCredits += Math.max(perf.audience - 30, 0);
    // add extra credit for every ten comedy attendees
    if ("comedy" === play.type) volumeCredits += Math.floor(perf.audier

    // print line for this order
    result += `  ${play.name}: ${format(thisAmount/100)} (${perf.audier
    totalAmount += thisAmount;
  }
  result += `Amount owed is ${format(totalAmount/100)}\n`;
  result += `You earned ${volumeCredits} credits\n`;
  return result;
}
```

Running that code on the test data files above results in the following output:

```
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
```

## Comments on the Starting Program

What are your thoughts on the design of this program? The first thing I'd say is that it's tolerable as it is—a program so short doesn't require any deep structure to be comprehensible. But remember my earlier point that I have to keep examples small. Imagine this program on a larger scale—perhaps hundreds of lines long. At that size, a single inline function is hard to understand.

Given that the program works, isn't any statement about its structure merely an aesthetic judgment, a dislike of "ugly" code? After all, the compiler doesn't care whether the code i ugly or clean. But when I change the system, there is a human involved, and humans do care. A poorly designed system is hard to change—because it is difficult to figure out wha to change and how these changes will interact with the existing code to get the behavior want. And if it is hard to figure out what to change, there is a good chance that I will make mistakes and introduce bugs.

Thus, if I'm faced with modifying a program with hundreds of lines of code, I'd rather it be structured into a set of functions and other program elements that allow me to understan more easily what the program is doing. If the program lacks structure, it's usually easier f me to add structure to the program first, and then make the change I need.

*When you have to add a feature to a program but the code is not structured in a convenient way, first refactor the program to make it easy to add the feature, then add the feature.*

In this case, I have a couple of changes that the users would like to make. First, they wan a statement printed in HTML. Consider what impact this change would have. I'm faced with adding conditional statements around every statement that adds a string to the resul

That will add a host of complexity to the function. Faced with that, most people prefer to copy the method and change it to emit HTML. Making a copy may not seem too onerous task, but it sets up all sorts of problems for the future. Any changes to the charging logic would force me to update both methods—and to ensure they are updated consistently. If I'm writing a program that will never change again, this kind of copy-and-paste is fine. Bu if it's a long-lived program, then duplication is a menace.

This brings me to a second change. The players are looking to perform more kinds of plays: they hope to add history, pastoral, pastoral-comical, historical-pastoral, tragical-historical, tragical-comical-historical-pastoral, scene individable, and poem unlimited to their repertoire. They haven't exactly decided yet what they want to do and when. This change will affect both the way their plays are charged for and the way volume credits ar calculated. As an experienced developer I can be sure that whatever scheme they come up with, they will change it again within six months. After all, when feature requests come they come not as single spies but in battalions.

Again, that `statement` method is where the changes need to be made to deal with changes in classification and charging rules. But if I copy `statement` to `htmlStatement`, I'd need to ensure that any changes are consistent. Furthermore, as the rules grow in complexity, it’s going to be harder to figure out where to make the changes and harder to do them without making a mistake.

Let me stress that it's these changes that drive the need to perform refactoring. If the cod works and doesn't ever need to change, it's perfectly fine to leave it alone. It would be nic to improve it, but unless someone needs to understand it, it isn't causing any real harm. Yet as soon as someone does need to understand how that code works, and struggles to follow it, then you have to do something about it.

## The First Step in Refactoring

Whenever I do refactoring, the first step is always the same. I need to ensure I have a solid set of tests for that section of code. The tests are essential because even though I will follow refactorings structured to avoid most of the opportunities for introducing bugs, I’m still human and still make mistakes. The larger a program, the more likely it is that my changes will cause something to break inadvertently—in the digital age, frailty's name is software.

Since the `statement` returns a string, what I do is create a few invoices, give each invoice a few performances of various kinds of plays, and generate the statement strings then do a string comparison between the new string and some reference strings that I have hand-checked. I set up all of these tests using a testing framework so I can run ther with just a simple keystroke in my development environment. The tests take only a few seconds to run, and as you will see, I run them often.

An important part of the tests is the way they report their results. They either go green, meaning that all the strings are identical to the reference strings, or red, showing a list of failures—the lines that turned out differently. The tests are thus self-checking. It is vital to make tests self-checking. If I don’t, I'd end up spending time hand-checking values from the test against values on a desk pad, and that would slow me down. Modern testing frameworks provide all the features needed to write and run self-checking tests.

*Before you start refactoring, make sure you have a solid suite of*

*Before you start refactoring, make sure you have a solid suite of tests. These tests must be self-checking.*

As I do the refactoring, I'll lean on the tests. I think of them as a bug detector to protect m against my own mistakes. By writing what I want twice, in the code and in the test, I have to make the mistake consistently in both places to fool the detector. By double-checking my work, I reduce the chance of doing something wrong. Although it takes time to build the tests, I end up saving that time, with considerable interest, by spending less time debugging. This is such an important part of refactoring that I devote a full chapter to it.

## Decomposing the `statement` Function

When refactoring a long function like this, I mentally try to identify points that separate different parts of the overall behavior. The first chunk that leaps to my eye is the switch statement in the middle.

```
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}`);
    }

    // add volume credits
    volumeCredits += Math.max(perf.audience - 30, 0);
    // add extra credit for every ten comedy attendees
    if ("comedy" === play.type) volumeCredits += Math.floor(perf.audience

    // print line for this order
    result += `  ${play.name}: ${format(thisAmount/100)} (${perf.audience
    totalAmount += thisAmount;
  }
```

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

As I look at this chunk, I conclude that it's calculating the charge for one performance. That conclusion is a piece of insight about the code. But as Ward Cunningham puts it, thi understanding is in my head—a notoriously volatile form of storage. I need to persist it by moving it from my head back into the code itself. That way, should I come back to it later, the code will tell me what it's doing—I don't have to figure it out again.

The way to put that understanding into code is to turn that chunk of code into its own function, naming it after what it does—something like `amountFor(aPerformance)`. When I want to turn a chunk of code into a function like this, I have a procedure for doing that minimizes my chances of getting it wrong. I wrote down this procedure and, to make easy to reference, named it Extract Function.

First, I need to look in the fragment for any variables that will no longer be in scope once I've extracted the code into its own function. In this case, I have three: `perf`, `play`, and `thisAmount`. The first two are used by the extracted code, but not modified, so I can pass them in as parameters. Modified variables need more care. Here, there is only one, so I can return it. I can also bring its initialization inside the extracted code. All of which yields this:

*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;
  }
```

> When I use a header like "*function someName...*" in italics for some code, that means that the following code is within the scope of the function, file, or class named in the header. There is usually other code with that scope that I won't show, as I'm not discussing it at the moment.

The original `statement` code now calls this function to populate `thisAmount`:

*top level...*

```
  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);

    // add volume credits
    volumeCredits += Math.max(perf.audience - 30, 0);
    // add extra credit for every ten comedy attendees
    if ("comedy" === play.type) volumeCredits += Math.floor(perf.audier

    // print line for this order
    result += `  ${play.name}: ${format(thisAmount/100)} (${perf.audier
    totalAmount += thisAmount;
  }
  result += `Amount owed is ${format(totalAmount/100)}\n`;
  result += `You earned ${volumeCredits} credits\n`;
  return result;
```

Once I've made this change, I immediately compile and test to see if I've broken anything It's an important habit to test after every refactoring, however simple. Mistakes are easy t make—at least, I find them easy to make. Testing after each change means that when I make a mistake, I only have a small change to consider in order to spot the error, which makes it far easier to find and fix. This is the essence of the refactoring process: small changes and testing after each change. If I try to do too much, making a mistake will forc me into a tricky debugging episode that can take a long time. Small changes, enabling a tight feedback loop, are the key to avoiding that mess.

> I use *compile* here to mean doing whatever is needed to make the JavaScript executable. Since JavaScrip is directly executable, that may mean nothing, but in other cases it may mean moving code to an output directory and/or using a processor such as Babel.

*Refactoring changes the programs in small steps, so if you make a mistake, it is easy to find where the bug is.*

This being JavaScript, I can extract `amountFor` into a nested function of `statement`. This is helpful as it means I don't have to pass data that's inside the scope of the containing function to the newly extracted function. That doesn't make a difference in this case, but it's one less issue to deal with.

In this case the tests passed, so my next step is to commit the change to my local versio control system. I use a version control system, such as git or mercurial, that allows me to make private commits. I commit after each successful refactoring, so I can easily get bac to a working state should I mess up later. I then squash changes into more significant commits before I push the changes to a shared repository.

Extract Function is a common refactoring to automate. If I was programming in Java, I would have instinctively reached for the key sequence for my IDE to perform this refactoring. As I write this, there is no such robust support for this refactoring in JavaScri tools, so I have to do this manually. It's not hard, although I have to be careful with those locally scoped variables.

Once I've used Extract Function, I take a look at what I've extracted to see if there are an quick and easy things I can do to clarify the extracted function. The first thing I do is rename some of the variables to make them clearer, such as changing `thisAmount` to `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;
  }
```

It's my coding standard to always call the return value from a function "result". That way I always know its role. Again, I compile, test, and commit. Then I move onto the first argument.

*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;
  }
```

Again, this is following my coding style. With a dynamically typed language such as JavaScript, it's useful to keep track of types—hence, my default name for a parameter includes the type name. I use an indefinite article with it unless there is some specific role information to capture in the name. I learned this convention from Kent Beck and continu to find it helpful.

*Any fool can write code that a computer can understand. Good programmers write code that humans can understand.*

Is this renaming worth the effort? Absolutely. Good code should clearly communicate wha it is doing, and variable names are a key to clear code. Never be afraid to change names to improve clarity. With good find-and-replace tools, it is usually not difficult; testing, and

static typing in a language that supports it, will highlight any occurrences you miss. And with automated refactoring tools, it's trivial to rename even widely used functions.

The next item to consider for renaming is the `play` parameter, but I have a different fate for that.

## Removing the `play` Variable

As I consider the parameters to `amountFor`, I look to see where they come from. `aPerformance` comes from the loop variable, so naturally changes with each iterationthrough the loop. But `play` is computed from the performance, so there's no nee to pass it in as a parameter at all—I can just recalculate it within `amountFor`. When I'm breaking down a long function, I like to get rid of variables like `play`, because temporary variables create a lot of locally scoped names that complicate extractions. The refactorin I will use here is Replace Temp with Query.

I begin by extracting the right-hand side of the assignment into a function.

*function statement…*

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

*top level…*

```
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);

    // add volume credits
    volumeCredits += Math.max(perf.audience - 30, 0);
    // add extra credit for every ten comedy attendees
    if ("comedy" === play.type) volumeCredits += Math.floor(perf.audie

    // print line for this order
    result += `  ${play.name}: ${format(thisAmount/100)} (${perf.audie
    totalAmount += thisAmount;
  }
  result += `Amount owed is ${format(totalAmount/100)}\n`;
  result += `You earned ${volumeCredits} credits\n`;
  return result;
```

I compile-test-commit, and then use Inline Variable.

*top level…*

```
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));
```

```
    // add volume credits
    volumeCredits += Math.max(perf.audience - 30, 0);
    // add extra credit for every ten comedy attendees
    if ("comedy" === playFor(perf).type) volumeCredits += Math.floor(pe

    // print line for this order
    result += `  ${playFor(perf).name}: ${format(thisAmount/100)} (${pe
    totalAmount += thisAmount;
  }
  result += `Amount owed is ${format(totalAmount/100)}\n`;
  result += `You earned ${volumeCredits} credits\n`;
  return result;
```

I compile-test-commit. With that inlined, I can then apply Change Function Declaration to `amountFor` to remove the `play` parameter. I do this in two steps. First, I use the new function inside `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;
}
```

I compile-test-commit, and then delete the parameter.

*top level...*

```
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)~~);

    // add volume credits
    volumeCredits += Math.max(perf.audience - 30, 0);
    // add extra credit for every ten comedy attendees
    if ("comedy" === playFor(perf).type) volumeCredits += Math.floor(pe

    // print line for this order
    result += `  ${playFor(perf).name}: ${format(thisAmount/100)} (${pe
    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;
  }
```

And compile-test-commit again.

This refactoring alarms some programmers. Previously, the code to look up the play was executed once in each loop iteration; now, it's executed thrice. I'll talk about the interplay of refactoring and performance later, but for the moment I'll just observe that this change unlikely to significantly affect performance, and even if it were, it is much easier to improv the performance of a well-factored code base.

The great benefit of removing local variables is that it makes it much easier to do extractions, since there is less local scope to deal with. Indeed, usually I'll take out local variables before I do any extractions.

Now that I'm done with the arguments to `amountFor`, I look back at where it's called. It's being used to set a temporary variable that's not updated again, so I apply Inline Variable

*top level...*

```
  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) {

      // add volume credits
      volumeCredits += Math.max(perf.audience - 30, 0);
      // add extra credit for every ten comedy attendees
      if ("comedy" === playFor(perf).type) volumeCredits += Math.floor(pe

      // print line for this order
      result += `  ${playFor(perf).name}: ${format(amountFor(perf)/100)}
      totalAmount += amountFor(perf);
    }
    result += `Amount owed is ${format(totalAmount/100)}\n`;
```

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

## Extracting Volume Credits

Here's the current state of the `statement` function body:

*top level...*

```
  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) {

      // add volume credits
      volumeCredits += Math.max(perf.audience - 30, 0);
      // add extra credit for every ten comedy attendees
      if ("comedy" === playFor(perf).type) volumeCredits += Math.floor(pe

      // print line for this order
      result += `  ${playFor(perf).name}: ${format(amountFor(perf)/100)}
      totalAmount += amountFor(perf);
    }
    result += `Amount owed is ${format(totalAmount/100)}\n`;
    result += `You earned ${volumeCredits} credits\n`;
    return result;
```

Now I get the benefit from removing the `play` variable as it makes it easier to extract the volume credits calculation by removing one of the locally scoped variables.

I still have to deal with the other two. Again, `perf` is easy to pass in, but `volumeCredit` is a bit more tricky as it is an accumulator updated in each pass of the loop. So my best bet is to initialize a shadow of it inside the extracted function and return it.

*function statement...*

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

*top level...*

```
  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);

      // print line for this order
      result += `  ${playFor(perf).name}: ${format(amountFor(perf)/100)}
      totalAmount += amountFor(perf);
    }
    result += `Amount owed is ${format(totalAmount/100)}\n`;
```

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

I remove the unnecessary (and, in this case, downright misleading) comment.

I compile-test-commit that, and then rename the variables inside the new function.

*function statement…*

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

I've shown it in one step, but as before I did the renames one at a time, with a compile-test-commit after each.

## Removing the `format` Variable

Let's look at the main `statement` method again:

*top level…*

```
  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);

      // print line for this order
      result += `  ${playFor(perf).name}: ${format(amountFor(perf)/100)}
      totalAmount += amountFor(perf);
    }
    result += `Amount owed is ${format(totalAmount/100)}\n`;
    result += `You earned ${volumeCredits} credits\n`;
    return result;
```

As I suggested before, temporary variables can be a problem. They are only useful within their own routine, and therefore they encourage long, complex routines. My next move, then, is to replace some of them. The easiest one is `format`. This is a case of assigning function to a temp, which I prefer to replace with a declared function.

*function statement…*

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

*top level…*

```
  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);

      // print line for this order
```

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

> Although changing a function variable to a declared function is a refactoring, I haven't named it and include it in the catalog. There are many refactorings that I didn't feel important enough for that. This one is both simple to do and relatively rare, so I didn't think it was worthwhile.

I'm not keen on the name—"format" doesn't really convey enough of what it's doing. "formatAsUSD" would be a bit too long-winded since it's being used in a string template, particularly within this small scope. I think the fact that it's formatting a currency amount i the thing to highlight here, so I pick a name that suggests that and apply Change Functio Declaration.

*top level…*

```
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);

    // print line for this order
    result += `  ${playFor(perf).name}: ${usd(amountFor(perf))} (${perf
    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
}
```

Naming is both important and tricky. Breaking a large function into smaller ones only add value if the names are good. With good names, I don't have to read the body of the function to see what it does. But it's hard to get names right the first time, so I use the be name I can think of for the moment, and don't hesitate to rename it later. Often, it takes a second pass through some code to realize what the best name really is.

As I'm changing the name, I also move the duplicated division by 100 into the function. Storing money as integer cents is a common approach—it avoids the dangers of storing fractional monetary values as floats but allows me to use arithmetic operators. Whenever want to display such a penny-integer number, however, I need a decimal, so my formattir function should take care of the division.

## Removing Total Volume Credits

My next target variable is `volumeCredits`. This is a trickier case, as it's built up during the iterations of the loop. My first move, then, is to use Split Loop to separate the accumulation of `volumeCredits`.

*top level…*

```
function statement (invoice, plays) {
  let totalAmount = 0;
  let volumeCredits = 0;
```

```
  let result = `Statement for ${invoice.customer}\n`;

  for (let perf of invoice.performances) {

    // print line for this order
    result += `  ${playFor(perf).name}: ${usd(amountFor(perf))} (${perf
    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;
```

With that done, I can use Slide Statements to move the declaration of the variable next to the loop.

*top level…*

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

    // print line for this order
    result += `  ${playFor(perf).name}: ${usd(amountFor(perf))} (${perf
    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;
```

Gathering together everything that updates the `volumeCredits` variable makes it easier to do Replace Temp with Query. As before, the first step is to apply Extract Function to the overall calculation of the variable.

*function statement…*

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

*top level…*

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

    // print line for this order
    result += `  ${playFor(perf).name}: ${usd(amountFor(perf))} (${perf
    totalAmount += amountFor(perf);
  }
  let volumeCredits = totalVolumeCredits();
  result += `Amount owed is ${usd(totalAmount)}\n`;
```

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

Once everything is extracted, I can apply Inline Variable:

*top level...*

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

      // print line for this order
      result += `  ${playFor(perf).name}: ${usd(amountFor(perf))} (${perf
      totalAmount += amountFor(perf);
    }

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

Let me pause for a bit to talk about what I've just done here. Firstly, I know readers will again be worrying about performance with this change, as many people are wary of repeating a loop. But most of the time, rerunning a loop like this has a negligible effect or performance. If you timed the code before and after this refactoring, you would probably not notice any significant change in speed—and that's usually the case. Most programmers, even experienced ones, are poor judges of how code actually performs. Many of our intuitions are broken by clever compilers, modern caching techniques, and the like. The performance of software usually depends on just a few parts of the code, ar changes anywhere else don't make an appreciable difference.

But "mostly" isn't the same as "alwaysly." Sometimes a refactoring will have a significant performance implication. Even then, I usually go ahead and do it, because it's much easi to tune the performance of well-factored code. If I introduce a significant performance issue during refactoring, I spend time on performance tuning afterwards. It may be that this leads to reversing some of the refactoring I did earlier—but most of the time, due to the refactoring, I can apply a more effective performance-tuning enhancement instead. I end up with code that's both clearer and faster.

So, my overall advice on performance with refactoring is: Most of the time you should ignore it. If your refactoring introduces performance slow-downs, finish refactoring first ar do performance tuning afterwards.

The second aspect I want to call your attention to is how small the steps were to remove `volumeCredits`. Here are the four steps, each followed by compiling, testing, and committing to my local source code repository:

- Split Loop to isolate the accumulation
- Slide Statements to bring the initializing code next to the accumulation
- Extract Function to create a function for calculating the total
- Inline Variable to remove the variable completely

I confess I don't always take quite as short steps as these—but whenever things get difficult, my first reaction is to take shorter steps. In particular, should a test fail during a refactoring, if I can't immediately see and fix the problem, I'll revert to my last good comm and redo what I just did with smaller steps. That works because I commit so frequently and because small steps are the key to moving quickly, particularly when working with difficult code.

I then repeat that sequence to remove `totalAmount`. I start by splitting the loop (compile-test-commit), then I slide the variable initialization (compile-test-commit), and

then I extract the function. There is a wrinkle here: The best name for the function is "totalAmount", but that's the name of the variable, and I can't have both at the same time So I give the new function a random name when I extract it (and compile-test-commit).

*function statement...*

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

*top level...*

```
  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
    }
    let totalAmount = appleSauce();

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

Then I inline the variable (compile-test-commit) and rename the function to something more sensible (compile-test-commit).

*top level...*

```
  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
    }
    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;
  }
```

I also take the opportunity to change the names inside my extracted functions to adhere t my convention.

*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: Lots of Nested Functions

Now is a good time to pause and take a look at the overall state of the code:

```
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
  }
  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/10
  }
  function volumeCreditsFor(aPerformance) {
    let result = 0;
    result += Math.max(aPerformance.audience - 30, 0);
    if ("comedy" === playFor(aPerformance).type) result += Math.floor(a
    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;
    }
  }
```

The structure of the code is much better now. The top-level `statement` function is now just seven lines of code, and all it does is laying out the printing of the statement. All the calculation logic has been moved out to a handful of supporting functions. This makes it easier to understand each individual calculation as well as the overall flow of the report.

## Splitting the Phases of Calculation and Formatting

So far, my refactoring has focused on adding enough structure to the function so that I ca understand it and see it in terms of its logical parts. This is often the case early in refactoring. Breaking down complicated chunks into small pieces is important, as is naming things well. Now, I can begin to focus more on the functionality change I want to make—specifically, providing an HTML version of this statement. In many ways, it's now much easier to do. With all the calculation code split out, all I have to do is write an HTMl version of the seven lines of code at the top. The problem is that these broken-out functions are nested within the textual statement method, and I don't want to copy and paste them into a new function, however well organized. I want the same calculation functions to be used by the text and HTML versions of the statement.

There are various ways to do this, but one of my favorite techniques is Split Phase. My aim here is to divide the logic into two parts: one that calculates the data required for the statement, the other that renders it into text or HTML. The first phase creates an intermediate data structure that it passes to the second.

I start a Split Phase by applying Extract Function to the code that makes up the second phase. In this case, that's the statement printing code, which is in fact the entire content o `statement`. This, together with all the nested functions, goes into its own top-level function which I call `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.a
  }
  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) {...}
```

I do my usual compile-test-commit, then create an object that will act as my intermediate data structure between the two phases. I pass this data object in as an argument to `renderPlainText` (compile-test-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.a
  }
  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) {...}
```

I now examine the other arguments used by `renderPlainText`. I want to move the dat that comes from them into the intermediate data structure, so that all the calculation code moves into the `statement` function and `renderPlainText` operates solely on data passed to it through the `data` parameter.

My first move is to take the customer and add it to the intermediate object (compile-test-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.a
  }
  result += `Amount owed is ${usd(totalAmount())}\n`;
  result += `You earned ${totalVolumeCredits()} credits\n`;
  return result;
```

Similarly, I add the performances, which allows me to delete the `invoice`parameter to `renderPlainText` (compile-test-commit).

*top level…*

```
  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
  }
  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;
}
```

Now I'd like the play name to come from the intermediate data. To do this, I need to enric the performance record with data from the play (compile-test-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;
  }
```

At the moment, I'm just making a copy of the performance object, but I'll shortly add data to this new record. I take a copy because I don't want to modify the data passed into the function. I prefer to treat data as immutable as much as I can—mutable state quickly becomes something rotten.

> The idiom `result = Object.assign({}, aPerformance)` looks very odd to people unfamiliar to JavaScript. It performs a shallow copy. I'd prefer to have a function for this, but it's one of those cases whe the idiom is so baked into JavaScript usage that writing my own function would look out of place for JavaScript programmers.

Now I have a spot for the play, I need to add it. To do that, I need to apply Move Function to `playFor` and `statement` (compile-test-commit).

*function statement...*

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

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

I then replace all the references to `playFor` in `renderPlainText` to use the data instead (compile-test-commit).

*function renderPlainText...*

```
  let result = `Statement for ${data.customer}\n`;
  for (let perf of data.performances) {
    result += `  ${perf.play.name}: ${usd(amountFor(perf))} (${perf.au
  }
  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(aPerfo
  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;
}
```

I then move `amountFor` in a similar way (compile-test-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.audienc
  }
  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;
  }
```

Next, I move the volume credits calculation (compile-test-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;
  }
```

Finally, I move the two calculations of the totals.

*function statement...*

```
    const statementData = {};
    statementData.customer = invoice.customer;
    statementData.performances = invoice.performances.map(enrichPerformar
    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.audienc
    }
    result += `Amount owed is ${usd(data.totalAmount)}\n`;
    result += `You earned ${data.totalVolumeCredits} credits\n`;
    return result;
```

Although I could have modified the bodies of these totals functions to use the `statementData` variable (as it's within scope), I prefer to pass the explicit parameter.

And, once I'm done with compile-test-commit after the move, I can't resist a couple quick shots of Replace Loop with Pipeline.

*function renderPlainText...*

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

I now extract all the first-phase code into its own function (compile-test-commit).

*top level…*

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

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

Since it's clearly separate now, I move it to its own file (and alter the name of the returned result to match my usual convention).

*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) {...}
```

One final swing of compile-test-commit—and now it's easy to write an HTML version.

*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<
  return result;
}

function usd(aNumber) {...}
```

(I moved `usd` to the top level, so that `renderHtml` could use it.)

## Status: Separated into Two Files (and Phases)

This is a good moment to take stock again and think about where the code is now. I have two files of code.

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

*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(aPer
  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);
}
```

I have more code than I did when I started: 70 lines (not counting `htmlStatement`) as opposed to 44, mostly due to the extra wrapping involved in putting things in functions. If all else is equal, more code is bad—but rarely is all else equal. The extra code breaks up the logic into identifiable parts, separating the calculations of the statements from the layout. This modularity makes it easier for me to understand the parts of the code and ho they fit together. Brevity is the soul of wit, but clarity is the soul of evolvable software. Adding this modularity allows to me to support the HTML version of the code without any duplication of the calculations.

*When programming, follow the camping rule: Always leave the code base healthier than when you found it.*

There are more things I could do to simplify the printing logic, but this will do for the moment. I always have to strike a balance between all the refactorings I could do and adding new features. At the moment, most people underprioritize refactoring—but there still is a balance. My rule is a variation on the camping rule: Always leave the code base healthier than when you found it. It will never be perfect, but it should be better.

## Reorganizing the Calculations by Type

Now I'll turn my attention to the next feature change: supporting more categories of plays each with its own charging and volume credits calculations. At the moment, to make changes here I have to go into the calculation functions and edit the conditions in there. The `amountFor` function highlights the central role the type of play has in the choice of calculations—but conditional logic like this tends to decay as further modifications are made unless it's reinforced by more structural elements of the programming language.

There are various ways to introduce structure to make this explicit, but in this case a natural approach is type polymorphism—a prominent feature of classical object-orientation. Classical OO has long been a controversial feature in the JavaScript world, but the ECMAScript 2015 version provides a sound syntax and structure for it. So it make sense to use it in a right situation—like this one.

My overall plan is to set up an inheritance hierarchy with comedy and tragedy subclasses that contain the calculation logic for those cases. Callers call a polymorphic amount function that the language will dispatch to the different calculations for the comedies and tragedies. I'll make a similar structure for the volume credits calculation. To do this, I utiliz a couple of refactorings. The core refactoring is Replace Conditional with Polymorphism, which changes a hunk of conditional code with polymorphism. But before I can do Replace Conditional with Polymorphism, I need to create an inheritance structure of som kind. I need to create a class to host the amount and volume credit functions.

I begin by reviewing the calculation code. (One of the pleasant consequences of the previous refactoring is that I can now ignore the formatting code, so long as I produce the same output data structure. I can further support this by adding tests that probe the intermediate data structure.)

*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(aPer
      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);
    }
```

## Creating a Performance Calculator

The `enrichPerformance` function is the key, since it populates the intermediate data structure with the data for each performance. Currently, it calls the conditional functions f amount and volume credits. What I need it to do is call those functions on a host class. Since that class hosts functions for calculating data about performances, I'll call it a 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;
  }
```

*top level...*

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

So far, this new object isn't doing anything. I want to move behavior into it—and I'd like tc start with the simplest thing to move, which is the play record. Strictly, I don't need to do this, as it's not varying polymorphically, but this way I'll keep all the data transforms in on place, and that consistency will make the code clearer.

To make this work, I will use Change Function Declaration to pass the performance's pla into the calculator.

*function createStatementData...*

```
  function enrichPerformance(aPerformance) {
    const calculator = new PerformanceCalculator(aPerformance, playFor(a
    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;
    }
  }
```

(I'm not saying compile-test-commit all the time any more, as I suspect you're getting tire of reading it. But I still do it at every opportunity. I do sometimes get tired of doing it—and give mistakes the chance to bite me. Then I learn and get back into the rhythm.)

## Moving Functions into the Calculator

The next bit of logic I move is rather more substantial for calculating the amount for a performance. I've moved functions around casually while rearranging nested functions—but this is a deeper change in the context of the function, so I'll step through the Move Function refactoring. The first part of this refactoring is to copy the logic over to its new context—the calculator class. Then, I adjust the code to fit into its new home, changing `aPerformance` to `this.performance` and `playFor(aPerformance)` to `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;
  }
```

I can compile at this point to check for any compile-time errors. "Compiling" in my development environment occurs as I execute the code, so what I actually do is run Babe That will be enough to catch any syntax errors in the new function—but little more than that. Even so, that can be a useful step.

Once the new function fits its home, I take the original function and turn it into a delegatir function so it calls the new function.

*function createStatementData…*

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

Now I can compile-test-commit to ensure the code is working properly in its new home. With that done, I use Inline Function to call the new function directly (compile-test-commit).

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

I repeat the same process to move the volume credits calculation.

*function createStatementData…*
```
  function enrichPerformance(aPerformance) {
    const calculator = new PerformanceCalculator(aPerformance, playFor(aI
    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.performanc
    return result;
  }
```

## Making the Performance Calculator Polymorphic

Now that I have the logic in a class, it's time to apply the polymorphism. The first step is t use Replace Type Code with Subclasses to introduce subclasses instead of the type code. For this, I need to create subclasses of the performance calculator and use the appropriate subclass in `createPerformanceData`. In order to get the right subclass, I need to replace the constructor call with a function, since JavaScript constructors can't return subclasses. So I use Replace Constructor with Factory Function.

*function createStatementData…*
```
  function enrichPerformance(aPerformance) {
    const calculator = createPerformanceCalculator(aPerformance, playFor(
    const result = Object.assign({}, aPerformance);
    result.play = calculator.play;
    result.amount = calculator.amount;
    result.volumeCredits = calculator.volumeCredits;
    return result;
  }
```

*top level…*
```
  function createPerformanceCalculator(aPerformance, aPlay) {
    return new PerformanceCalculator(aPerformance, aPlay);
  }
```

With that now a function, I can create subclasses of the performance calculator and get the creation function to select which one to return.

*top level…*

```
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 {
}
```

This sets up the structure for the polymorphism, so I can now move on to Replace Conditional with Polymorphism.

I start with the calculation of the amount for tragedies.

*class TragedyCalculator…*

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

Just having this method in the subclass is enough to override the superclass conditional. But if you're as paranoid as I am, you might do this:

*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;
}
```

> I could have removed the case for tragedy and let the default branch throw an error. But I like the explicit throw—and it will only be there for a couple more minutes (which is why I threw a string, not a better error object).

After a compile-test-commit of that, I move the comedy case down too.

*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;
  }
```

I can now remove the superclass `amount` method, as it should never be called. But it's kinder to my future self to leave a tombstone.

*class PerformanceCalculator…*

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

The next conditional to replace is the volume credits calculation. Looking at the discussio of future categories of plays, I notice that most plays expect to check if audience is above 30, with only some categories introducing a variation. So it makes sense to leave the mo common case on the superclass as a default, and let the variations override it as necessary. So I just push down the case for comedies:

*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: Creating the Data with the Polymorphic Calculator

Time to reflect on what introducing the polymorphic calculator did to the code.

*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, playFo
      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 /
  }
}
```

Again, the code has increased in size as I've introduced structure. The benefit here is tha the calculations for each kind of play are grouped together. If most of the changes will be to this code, it will be helpful to have it clearly separated like this. Adding a new kind of play requires writing a new subclass and adding it to the creation function.

The example gives some insight as to when using subclasses like this is useful. Here, I'v moved the conditional lookup from two functions (`amountFor` and `volumeCreditsFor` to a single constructor function `createPerformanceCalculator`. The more functions there are that depend on the same type of polymorphism, the more useful this approach becomes.

An alternative to what I've done here would be to have `createPerformanceData` retur the calculator itself, instead of the calculator populating the intermediate data structure. One of the nice features of JavaScript's class system is that with it, using getters looks lik

regular data access. My choice on whether to return the instance or calculate separate output data depends on who is using the downstream data structure. In this case, I preferred to show how to use the intermediate data structure to hide the decision to use a polymorphic calculator.

## Final Thoughts

This is a simple example, but I hope it will give you a feeling for what refactoring is like. I've used several refactorings, including Extract Function, Inline Variable, Move Function, and Replace Conditional with Polymorphism.

There were three major stages to this refactoring episode: decomposing the original function into a set of nested functions, using Split Phase to separate the calculation and printing code, and finally introducing a polymorphic calculator for the calculation logic. Each of these added structure to the code, enabling me to better communicate what the code was doing.

As is often the case with refactoring, the early stages were mostly driven by trying to understand what was going on. A common sequence is: Read the code, gain some insight, and use refactoring to move that insight from your head back into the code. The clearer code then makes it easier to understand it, leading to deeper insights and a beneficial positive feedback loop. There are still some improvements I could make, but I feel I've done enough to pass my test of leaving the code significantly better than how I found it.

*The true test of good code is how easy it is to change it.*

I'm talking about improving the code—but programmers love to argue about what good code looks like. I know some people object to my preference for small, well-named functions. If we consider this to be a matter of aesthetics, where nothing is either good or bad but thinking makes it so, we lack any guide but personal taste. I believe, however, th we can go beyond taste and say that the true test of good code is how easy it is to chang it. Code should be obvious: When someone needs to make a change, they should be ab to find the code to be changed easily and to make the change quickly without introducing any errors. A healthy code base maximizes our productivity, allowing us to build more features for our users both faster and more cheaply. To keep code healthy, pay attention what is getting between the programming team and that ideal, then refactor to get closer to the ideal.

But the most important thing to learn from this example is the rhythm of refactoring. Whenever I've shown people how I refactor, they are surprised by how small my steps ar each step leaving the code in a working state that compiles and passes its tests. I was ju as surprised myself when Kent Beck showed me how to do this in a hotel room in Detroit two decades ago. The key to effective refactoring is recognizing that you go faster when you take tiny steps, the code is never broken, and you can compose those small steps into substantial changes. Remember that—and the rest is silence.

*previous:*
Preface

*next:*
Principles in Refactoring