Chapter 6

A First Set of Refactorings

I'm starting the catalog with a set of refactorings that I consider the most useful to learn first.

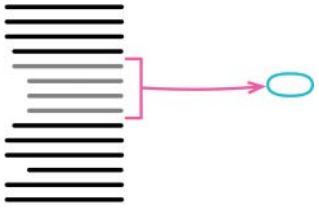
Probably the most common refactoring I do is extracting code into a function (*Extract Function (106)*) or a variable (*Extract Variable (119)*). Since refactoring is all about change, it's no surprise that I also frequently use the inverses of those two (*Inline Function (115)* and *Inline Variable (123)*).

Extraction is all about giving names, and I often need to change the names as I learn. *Change Function Declaration (124)* changes names of functions; I also use that refactoring to add or remove a function's arguments. For variables, I use *Rename Variable (137)*, which relies on *Encapsulate Variable (132)*. When changing function arguments, I often find it useful to combine a common clump of arguments into a single object with *Introduce Parameter Object (140)*.

Forming and naming functions are essential low-level refactorings—but, once created, it's necessary to group functions into higher-level modules. I use *Combine Functions into Class (144)* to group functions, together with the data they operate on, into a class. Another path I take is to combine them into a transform (*Combine Functions into Transform (149)*), which is particularly handy with read-only data. At a step further in scale, I can often form these modules into distinct processing phases using *Split Phase (154)*.

Extract Function

formerly: *Extract Method*
inverse of: *Inline Function (115)*



```
function printOwing(invoice) {
  printBanner();
  let outstanding  = calculateOutstanding();

  //print details
  console.log(`name: ${invoice.customer}`);
  console.log(`amount: ${outstanding}`);
}
```

```
function printOwing(invoice) {
  printBanner();
  let outstanding  = calculateOutstanding();
  printDetails(outstanding);

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

Motivation

Extract Function is one of the most common refactorings I do. (Here, I use the term "function" but the same is true for a method in an object-oriented language, or any kind of procedure or subroutine.) I look at a fragment of code, understand what it is doing, then extract it into its own function named after its purpose.

During my career, I've heard many arguments about when to enclose code in its own function. Some of these guidelines were based on length: Functions should be no larger than fit on a screen. Some were based on reuse: Any code

used more than once should be put in its own function, but code only used once should be left inline. The argument that makes most sense to me, however, is the separation between intention and implementation. If you have to spend effort looking at a fragment of code and figuring out *what* it's doing, then you should extract it into a function and name the function after the "what." Then, when you read it again, the purpose of the function leaps right out at you, and most of the time you won't need to care about how the function fulfills its purpose (which is the body of the function).

Once I accepted this principle, I developed a habit of writing very small functions—typically, only a few lines long. To me, any function with more than half-a-dozen lines of code starts to smell, and it's not unusual for me to have functions that are a single line of code. The fact that size isn't important was brought home to me by an example that Kent Beck showed me from the original Smalltalk system. Smalltalk in those days ran on black-and-white systems. If you wanted to highlight some text or graphics, you would reverse the video. Smalltalk's graphics class had a method for this called `highlight`, whose implementation was just a call to the method `reverse`. The name of the method was longer than its implementation—but that didn't matter because there was a big distance between the intention of the code and its implementation.

Some people are concerned about short functions because they worry about the performance cost of a function call. When I was young, that was occasionally a factor, but that's very rare now. Optimizing compilers often work better with shorter functions which can be cached more easily. As always, follow the general guidelines on performance optimization.

Small functions like this only work if the names are good, so you need to pay good attention to naming. This takes practice—but once you get good at it, this approach can make code remarkably self-documenting.

Often, I see fragments of code in a larger function that start with a comment to say what they do. The comment is often a good hint for the name of the function when I extract that fragment.

Mechanics

- Create a new function, and name it after the intent of the function (name it by what it does, not by how it does it).

  If the code I want to extract is very simple, such as a single function call, I still extract it if the name of the new function will reveal the intent of the code in a better way. If I can't come up with a more meaningful name, that's a sign that I shouldn't extract the code. However, I don't have to come up with the best name right away; sometimes a good name only appears as I work with the extraction. It's OK to extract a function, try to work with it, realize it isn't helping, and then inline it back again. As long as I've learned something, my time wasn't wasted.

If the language supports nested functions, nest the extracted function inside the source function. That will reduce the amount of out-of-scope variables to deal with after the next couple of steps. I can always use *Move Function (198)* later.

- Copy the extracted code from the source function into the new target function.
- Scan the extracted code for references to any variables that are local in scope to the source function and will not be in scope for the extracted function. Pass them as parameters.

  If I extract into a nested function of the source function, I don't run into these problems.

  Usually, these are local variables and parameters to the function. The most general approach is to pass all such parameters in as arguments. There are usually no difficulties for variables that are used but not assigned to.

  If a variable is only used inside the extracted code but is declared outside, move the declaration into the extracted code.

  Any variables that are assigned to need more care if they are passed by value. If there's only one of them, I try to treat the extracted code as a query and assign the result to the variable concerned.

  Sometimes, I find that too many local variables are being assigned by the extracted code. It's better to abandon the extraction at this point. When this happens, I consider other refactorings such as *Split Variable (240)* or *Replace Temp with Query (178)* to simplify variable usage and revisit the extraction later.

- Compile after all variables are dealt with.

  Once all the variables are dealt with, it can be useful to compile if the language environment does compile-time checks. Often, this will help find any variables that haven't been dealt with properly.

- Replace the extracted code in the source function with a call to the target function.
- Test.
- Look for other code that's the same or similar to the code just extracted, and consider using *Replace Inline Code with Function Call (222)* to call the new function.

  Some refactoring tools support this directly. Otherwise, it can be worth doing some quick searches to see if duplicate code exists elsewhere.

Example: No Variables Out of Scope

In the simplest case, Extract Function is trivially easy.

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

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

  // calculate outstanding
  for (const o of invoice.orders) {
    outstanding += o.amount;
  }

  // record due date
  const today = Clock.today;
  invoice.dueDate = new Date(today.getFullYear(), today.getMonth(), today.getDate() + 30);

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

You may be wondering what the `Clock.today` is about. It is a Clock Wrapper [mf-cw]—an object that wraps calls to the system clock. I avoid putting direct calls to things like `Date.now()` in my code, because it leads to nondeterministic tests and makes it difficult to reproduce error conditions when diagnosing failures.

It's easy to extract the code that prints the banner. I just cut, paste, and put in a call:

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

  printBanner();

  // calculate outstanding
  for (const o of invoice.orders) {
    outstanding += o.amount;
  }

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

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

Similarly, I can take the printing of details and extract that too:

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

  printBanner();

  // calculate outstanding
  for (const o of invoice.orders) {
    outstanding += o.amount;
  }

  // record due date
  const today = Clock.today;
  invoice.dueDate = new Date(today.getFullYear(), today.getMonth(), today.getDate() + 30);

  printDetails();

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

This makes Extract Function seem like a trivially easy refactoring. But in many situations, it turns out to be rather more tricky.

In the case above, I defined `printDetails` so it was nested inside `printOwing`. That way it was able to access all the variables defined in `printOwing`. But that's not an option to me if I'm programming in a language that doesn't allow nested functions. Then I'm faced, essentially, with the problem of extracting the function to the top level, which means I have to pay attention to any variables that exist only in the scope of the source function. These are the arguments to the original function and the temporary variables defined in the function.

Example: Using Local Variables

The easiest case with local variables is when they are used but not reassigned. In this case, I can just pass them in as parameters. So if I have the following function:

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

  printBanner();

  // calculate outstanding
  for (const o of invoice.orders) {
    outstanding += o.amount;
  }

  // record due date
  const today = Clock.today;
  invoice.dueDate = new Date(today.getFullYear(), today.getMonth(), today.getDate() + 30);

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

I can extract the printing of details passing two parameters:

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

  printBanner();

  // calculate outstanding
  for (const o of invoice.orders) {
    outstanding += o.amount;
  }

  // record due date
  const today = Clock.today;
  invoice.dueDate = new Date(today.getFullYear(), today.getMonth(), today.getDate() + 30);

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

The same is true if the local variable is a structure (such as an array, record, or object) and I modify that structure. So, I can similarly extract the setting of the due date:

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

  printBanner();

  // calculate outstanding
  for (const o of invoice.orders) {
    outstanding += o.amount;
  }

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

Example: Reassigning a Local Variable

It's the assignment to local variables that becomes complicated. In this case, we're only talking about temps. If I see an assignment to a parameter, I immediately use *Split Variable (240)*, which turns it into a temp.

For temps that are assigned to, there are two cases. The simpler case is where the variable is a temporary variable used only within the extracted code. When that happens, the variable just exists within the extracted code. Sometimes, particularly when variables are initialized at some distance before they are used, it's handy to use *Slide Statements (223)* to get all the variable manipulation together.

The more awkward case is where the variable is used outside the extracted function. In that case, I need to return the new value. I can illustrate this with the following familiar-looking function:

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

  printBanner();

  // calculate outstanding
  for (const o of invoice.orders) {
    outstanding += o.amount;
  }

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

I've shown the previous refactorings all in one step, since they were straightforward, but this time I'll take it one step at a time from the mechanics. First, I'll slide the declaration next to its use.

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

  // calculate outstanding
  let outstanding = 0;
  for (const o of invoice.orders) {
    outstanding += o.amount;
  }

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

I then copy the code I want to extract into a target function.

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

  // calculate outstanding
  let outstanding = 0;
  for (const o of invoice.orders) {
    outstanding += o.amount;
  }

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

Since I moved the declaration of `outstanding` into the extracted code, I don't need to pass it in as a parameter. The `outstanding` variable is the only one reassigned in the extracted code, so I can return it.

My JavaScript environment doesn't yield any value by compiling—indeed less than I'm getting from the syntax analysis in my editor—so there's no step to do here. My next thing to do is to replace the original code with a call to the new function. Since I'm returning the value, I need to store it in the original variable.

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

Before I consider myself done, I rename the return value to follow my usual coding style.

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

I also take the opportunity to change the original `outstanding` *into a* `const`.

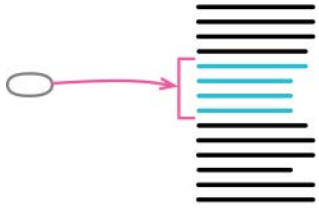
At this point you may be wondering, "What happens if more than one variable needs to be returned?"

Here, I have several options. Usually I prefer to pick different code to extract. I like a function to return one value, so I would try to arrange for multiple functions for the different values. If I really need to extract with multiple values, I can form a record and return that—but usually I find it better to rework the temporary variables instead. Here I like using *Replace Temp with Query (178)* and *Split Variable (240)*.

This raises an interesting question when I'm extracting functions that I expect to then move to another context, such as top level. I prefer small steps, so my instinct is to extract into a nested function first, then move that nested function to its new context. But the tricky part of this is dealing with variables and I don't expose that difficulty until I do the move. This argues that even though I can extract into a nested function, it makes sense to extract to at least the sibling level of the source function first, so I can immediately tell if the extracted code makes sense.

Inline Function

formerly: *Inline Method*
inverse of: *Extract Function (106)*



```
function getRating(driver) {
  return moreThanFiveLateDeliveries(driver) ? 2 : 1;
}

function moreThanFiveLateDeliveries(driver) {
  return driver.numberOfLateDeliveries > 5;
}
```

```
function getRating(driver) {
  return (driver.numberOfLateDeliveries > 5) ? 2 : 1;
}
```

Motivation

One of the themes of this book is using short functions named to show their intent, because these functions lead to clearer and easier to read code. But sometimes, I do come across a function in which the body is as clear as the name. Or, I refactor the body of the code into something that is just as clear as the name. When this happens, I get rid of the function. Indirection can be helpful, but needless indirection is irritating.

I also use Inline Function is when I have a group of functions that seem badly factored. I can inline them all into one big function and then reextract the functions the way I prefer.

I commonly use Inline Function when I see code that's using too much indirection—when it seems that every function does simple delegation to another function, and I get lost in all the delegation. Some of this indirection may be worthwhile, but not all of it. By inlining, I can flush out the useful ones and eliminate the rest.

Mechanics

- Check that this isn't a polymorphic method.

  If this is a method in a class, and has subclasses that override it, then I can't inline it.

- Find all the callers of the function.
- Replace each call with the function's body.
- Test after each replacement.

  The entire inlining doesn't have to be done all at once. If some parts of the inline are tricky, they can be done gradually as opportunity permits.

- Remove the function definition.

Written this way, Inline Function is simple. In general, it isn't. I could write pages on how to handle recursion, multiple return points, inlining a method into another object when you don't have accessors, and the like. The reason I don't is that if you encounter these complexities, you shouldn't do this refactoring.

Example

In the simplest case, this refactoring is so easy it's trivial. I start with

```
function rating(aDriver) {
  return moreThanFiveLateDeliveries(aDriver) ? 2 : 1;
}
function moreThanFiveLateDeliveries(aDriver) {
  return aDriver.numberOfLateDeliveries > 5;
}
```

I can just take the return expression of the called function and paste it into the caller to replace the call.

```
function rating(aDriver) {
  return aDriver.numberOfLateDeliveries > 5 ? 2 : 1;
}
```

But it can be a little more involved than that, requiring me to do more work to fit the code into its new home. Consider the case where I start with this slight variation on the earlier initial code.

```
function rating(aDriver) {
  return moreThanFiveLateDeliveries(aDriver) ? 2 : 1;
}

function moreThanFiveLateDeliveries(dvr) {
  return dvr.numberOfLateDeliveries > 5;
}
```

Almost the same, but now the declared argument on `moreThanFiveLateDeliveries` is different to the name of the passed-in argument. So I have to fit the code a little when I do the inline.

```
function rating(aDriver) {
  return aDriver.numberOfLateDeliveries > 5 ? 2 : 1;
}
```

It can be even more involved than this. Consider this code:

```
function reportLines(aCustomer) {
  const lines = [];
  gatherCustomerData(lines, aCustomer);
  return lines;
}
function gatherCustomerData(out, aCustomer) {
  out.push(["name", aCustomer.name]);
  out.push(["location", aCustomer.location]);
}
```

Inlining `gatherCustomerData` into `reportLines` isn't a simple cut and paste. It's not too complicated, and most times I would still do this in one go, with a bit of fitting. But to be cautious, it may make sense to move one line at a time. So I'd start with using *Move Statements to Callers (217)* on the first line (I'd do it the simple way with a cut, paste, and fit).

```
function reportLines(aCustomer) {
  const lines = [];
  lines.push(["name", aCustomer.name]);
  gatherCustomerData(lines, aCustomer);
  return lines;
}
function gatherCustomerData(out, aCustomer) {
  out.push(["name", aCustomer.name]);
  out.push(["location", aCustomer.location]);
}
```

I then continue with the other lines until I'm done.

```
function reportLines(aCustomer) {
  const lines = [];
  lines.push(["name", aCustomer.name]);
  lines.push(["location", aCustomer.location]);
  return lines;
}
```

The point here is to always be ready to take smaller steps. Most of the time, with the small functions I normally write, I can do Inline Function in one go, even if there is a bit of refitting to do. But if I run into complications, I go one line at a time. Even with one line, things can get a bit awkward; then, I'll use the more elaborate mechanics for *Move Statements to Callers (217)* to break things down even more. And if, feeling confident, I do something the quick way and the tests break, I prefer to revert back to my last green code and repeat the refactoring with smaller steps and a touch of chagrin.

Extract Variable

formerly: *Introduce Explaining Variable*
inverse of: *Inline Variable (123)*

```
return order.quantity * order.itemPrice -
  Math.max(0, order.quantity - 500) * order.itemPrice * 0.05 +
  Math.min(order.quantity * order.itemPrice * 0.1, 100);
```

```
const basePrice = order.quantity * order.itemPrice;
const quantityDiscount = Math.max(0, order.quantity - 500) * order.itemPrice * 0.05;
const shipping = Math.min(basePrice * 0.1, 100);
return basePrice - quantityDiscount + shipping;
```

Motivation

Expressions can become very complex and hard to read. In such situations, local variables may help break the expression down into something more manageable. In particular, they give me an ability to name a part of a more complex piece of logic. This allows me to better understand the purpose of what's happening.

Such variables are also handy for debugging, since they provide an easy hook for a debugger or print statement to capture.

If I'm considering Extract Variable, it means I want to add a name to an expression in my code. Once I've decided I want to do that, I also think about the context of that name. If it's only meaningful within the function I'm working on, then Extract Variable is a good choice—but if it makes sense in a broader context, I'll consider making the name available in that broader context, usually as a function. If the name is available more widely, then other code can use that expression without having to repeat the expression, leading to less duplication and a better statement of my intent.

The downside of promoting the name to a broader context is extra effort. If it's significantly more effort, I'm likely to leave it till later when I can use *Replace Temp with Query (178)*. But if it's easy, I like to do it now so the name is immediately available in the code. As a good example of this, if I'm working in a class, then *Extract Function (106)* is very easy to do.

Mechanics

- Ensure that the expression you want to extract does not have side effects.
- Declare an immutable variable. Set it to a copy of the expression you want to name.
- Replace the original expression with the new variable.
- Test.

If the expression appears more than once, replace each occurrence with the variable, testing after each replacement.

Example

I start with a simple calculation

```
function price(order) {
  //price is base price - quantity discount + shipping
  return order.quantity * order.itemPrice -
    Math.max(0, order.quantity - 500) * order.itemPrice * 0.05 +
    Math.min(order.quantity * order.itemPrice * 0.1, 100);
}
```

Simple as it may be, I can make it still easier to follow. First, I recognize that the base price is the multiple of the quantity and the item price.

```
function price(order) {
  //price is base price - quantity discount + shipping
  return order.quantity * order.itemPrice -
    Math.max(0, order.quantity - 500) * order.itemPrice * 0.05 +
    Math.min(order.quantity * order.itemPrice * 0.1, 100);
}
```

Once that understanding is in my head, I put it in the code by creating and naming a variable for it.

```
function price(order) {
  //price is base price - quantity discount + shipping
  const basePrice = order.quantity * order.itemPrice;
  return order.quantity * order.itemPrice -
    Math.max(0, order.quantity - 500) * order.itemPrice * 0.05 +
    Math.min(order.quantity * order.itemPrice * 0.1, 100);
}
```

Of course, just declaring and initializing a variable doesn't do anything; I also have to use it, so I replace the expression that I used as its source.

```
function price(order) {
  //price is base price - quantity discount + shipping
  const basePrice = order.quantity * order.itemPrice;
  return basePrice -
    Math.max(0, order.quantity - 500) * order.itemPrice * 0.05 +
    Math.min(order.quantity * order.itemPrice * 0.1, 100);
}
```

That same expression is used later on, so I can replace it with the variable there too.

```
function price(order) {
  //price is base price - quantity discount + shipping
  const basePrice = order.quantity * order.itemPrice;
  return basePrice -
    Math.max(0, order.quantity - 500) * order.itemPrice * 0.05 +
    Math.min(basePrice * 0.1, 100);
}
```

The next line is the quantity discount, so I can extract that too.

```
function price(order) {
  //price is base price - quantity discount + shipping
  const basePrice = order.quantity * order.itemPrice;
  const quantityDiscount = Math.max(0, order.quantity - 500) * order.itemPrice * 0.05;
  return basePrice -
    quantityDiscount +
    Math.min(basePrice * 0.1, 100);
}
```

Finally, I finish with the shipping. As I do that, I can remove the comment, too, because it no longer says anything the code doesn't say.

```
function price(order) {
  const basePrice = order.quantity * order.itemPrice;
  const quantityDiscount = Math.max(0, order.quantity - 500) * order.itemPrice * 0.05;
  const shipping = Math.min(basePrice * 0.1, 100);
  return basePrice - quantityDiscount + shipping;
}
```

Example: With a Class

Here's the same code, but this time in the context of a class:

```
class Order {
  constructor(aRecord) {
    this._data = aRecord;
  }
```

```
    get quantity()  {return this._data.quantity;}
    get itemPrice() {return this._data.itemPrice;}

    get price() {
      return this.quantity * this.itemPrice -
        Math.max(0, this.quantity - 500) * this.itemPrice * 0.05 +
        Math.min(this.quantity * this.itemPrice * 0.1, 100);
    }
  }
```

In this case, I want to extract the same names, but I realize that the names apply to the `Order` as a whole, not just the calculation of the price. Since they apply to the whole order, I'm inclined to extract the names as methods rather than variables.

```
class Order {
  constructor(aRecord) {
    this._data = aRecord;
  }
  get quantity()  {return this._data.quantity;}
  get itemPrice() {return this._data.itemPrice;}

  get price() {
    return this.basePrice - this.quantityDiscount + this.shipping;
  }
  get basePrice()        {return this.quantity * this.itemPrice;}
  get quantityDiscount() {return Math.max(0, this.quantity - 500) * this.itemPrice * 0.05;}
  get shipping()         {return Math.min(this.basePrice * 0.1, 100);}
}
```

This is one of the great benefits of objects—they give you a reasonable amount of context for logic to share other bits of logic and data. For something as simple as this, it doesn't matter so much, but with a larger class it becomes very useful to call out common hunks of behavior as their own abstractions with their own names to refer to them whenever I'm working with the object.

Inline Variable

formerly: *Inline Temp*
inverse of: *Extract Variable (119)*

```
let basePrice = anOrder.basePrice;
return (basePrice > 1000);
```

```
return anOrder.basePrice > 1000;
```

Motivation

Variables provide names for expressions within a function, and as such they are usually a Good Thing. But sometimes, the name doesn't really communicate more than the expression itself. At other times, you may find that a variable gets in the way of refactoring the neighboring code. In these cases, it can be useful to inline the variable.

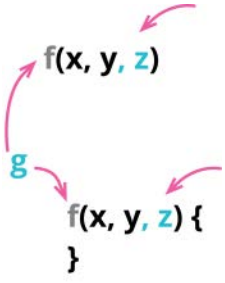
Mechanics

- Check that the right-hand side of the assignment is free of side effects.
- If the variable isn't already declared immutable, do so and test.

  This checks that it's only assigned to once.
- Find the first reference to the variable and replace it with the right-hand side of the assignment.
- Test.
- Repeat replacing references to the variable until you've replaced all of them.
- Remove the declaration and assignment of the variable.
- Test.

Change Function Declaration

aka: *Rename Function*
formerly: *Rename Method*
formerly: *Add Parameter*
formerly: *Remove Parameter*
aka: *Change Signature*




```
function circum(radius) {...}
```

```
function circumference(radius) {...}
```

Motivation

Functions represent the primary way we break a program down into parts. Function declarations represent how these parts fit together—effectively, they represent the joints in our software systems. And, as with any construction, much depends on those joints. Good joints allow me to add new parts to the system easily, but bad ones are a constant source of difficulty, making it harder to figure out what the software does and how to modify it as my needs change. Fortunately, software, being soft, allows me to change these joints, providing I do it carefully.

The most important element of such a joint is the name of the function. A good name allows me to understand what the function does when I see it called, without seeing the code that defines its implementation. However, coming up with good names is hard, and I rarely get my names right the first time. When I find a name that's confused me, I'm tempted to leave it—after all, it's only a name. This is the work of the evil demon *Obfuscatis*; for the sake of my program's soul I must never listen to him. If I see a function with the wrong name, it is imperative that I change it as soon as I understand what a better name could be. That way,

the next time I'm looking at this code, I don't have to figure out *again* what's going on. (Often, a good way to improve a name is to write a comment to describe the function's purpose, then turn that comment into a name.)

Similar logic applies to a function's parameters. The parameters of a function dictate how a function fits in with the rest of its world. Parameters set the context in which I can use a function. If I have a function to format a person's telephone number, and that function takes a person as its argument, then I can't use it to format a company's telephone number. If I replace the person parameter with the telephone number itself, then the formatting code is more widely useful.

Apart from increasing a function's range of applicability, I can also remove some coupling, changing what modules need to connect to others. Telephone formatting logic may sit in a module that has no knowledge about people. Reducing how much modules need to know about each other helps reduce how much I need to put into my brain when I change something—and my brain isn't as big as it used to be (that doesn't say anything about the size of its container, though).

Choosing the right parameters isn't something that adheres to simple rules. I may have a simple function for determining if a payment is overdue, by looking at if it's older than 30 days. Should the parameter to this function be the payment object, or the due date of the payment? Using the payment couples the function to the interface of the payment object. But if I use the payment, I can easily access other properties of the payment, should the logic evolve, without having to change every bit of code that calls this function—essentially, increasing the encapsulation of the function.

The only right answer to this puzzle is that there is no right answer, especially over time. So I find it's essential to be familiar with Change Function Declaration so the code can evolve with my understanding of what the best joints in the code need to be.

Usually, I only use the main name of a refactoring when I refer to it from elsewhere in this book. However, since renaming is such a significant use case for Change Function Declaration, if I'm just renaming something, I'll refer to this refactoring as *Rename Function* to make it clearer what I'm doing. Whether I'm merely renaming or manipulating the parameters, I use the same mechanics.

Mechanics

In most of the refactorings in this book, I present only a single set of mechanics. This isn't because there is only one set that will do the job but because, usually, one set of mechanics will work reasonably well for most cases. Change Function Declaration, however, is an exception. The simple mechanics are often effective, but there are plenty of cases when a more gradual migration makes more sense. So, with this refactoring, I look at the change and ask myself if I think I can change the declaration and all its callers easily in one go. If so, I follow the simple mechanics. The migration-style mechanics allow me to change the callers more gradually—which is important if I have lots of them, they are awkward to get

to, the function is a polymorphic method, or I have a more complicated change to the declaration.

Simple Mechanics

- If you're removing a parameter, ensure it isn't referenced in the body of the function.
- Change the method declaration to the desired declaration.
- Find all references to the old method declaration, update them to the new one.
- Test.

It's often best to separate changes, so if you want to both change the name and add a parameter, do these as separate steps. (In any case, if you run into trouble, revert and use the migration mechanics instead.)

Migration Mechanics

- If necessary, refactor the body of the function to make it easy to do the following extraction step.
- Use *Extract Function (106)* on the function body to create the new function.

  If the new function will have the same name as the old one, give the new function a temporary name that's easy to search for.
- If the extracted function needs additional parameters, use the simple mechanics to add them.
- Test.
- Apply *Inline Function (115)* to the old function.
- If you used a temporary name, use *Change Function Declaration (124)* again to restore it to the original name.
- Test.

If you're changing a method on a class with polymorphism, you'll need to add indirection for each binding. If the method is polymorphic within a single class hierarchy, you only need the forwarding method on the superclass. If the polymorphism has no superclass link, then you'll need forwarding methods on each implementation class.

If you are refactoring a published API, you can pause the refactoring once you've created the new function. During this pause, deprecate the original function and wait for clients to change to the new function. The original function declara-

tion can be removed when (and if) you're confident all the clients of the old function have migrated to the new one.

Example: Renaming a Function (Simple Mechanics)

Consider this function with an overly abbreved name:

```
function circum(radius) {
  return 2 * Math.PI * radius;
}
```

I want to change that to something more sensible. I begin by changing the declaration:

```
function circumference(radius) {
  return 2 * Math.PI * radius;
}
```

I then find all the callers of `circum` and change the name to `circumference`.

Different language environments have an impact on how easy it is to find all the references to the old function. Static typing and a good IDE provide the best experience, usually allowing me to rename functions automatically with little chance of error. Without static typing, this can be more involved; even good searching tools will then have a lot of false positives.

I use the same approach for adding or removing parameters: find all the callers, change the declaration, and change the callers. It's often better to do these as separate steps—so, if I'm both renaming the function and adding a parameter, I first do the rename, test, then add the parameter, and test again.

A disadvantage of this simple way of doing the refactoring is that I have to do all the callers and the declaration (or all of them, if polymorphic) at once. If there are only a few of them, or if I have decent automated refactoring tools, this is reasonable. But if there's a lot, it can get tricky. Another problem is when the names aren't unique—e.g., I want to rename the a `changeAddress` method on a person class but the same method, which I don't want to change, exists on an insurance agreement class. The more complex the change is, the less I want to do it in one go like this. When this kind of problem arises, I use the migration mechanics instead. Similarly, if I use simple mechanics and something goes wrong, I'll revert the code to the last known good state and try again using migration mechanics.

Example: Renaming a Function (Migration Mechanics)

Again, I have this function with its overly abbreved name:

```
function circum(radius) {
  return 2 * Math.PI * radius;
}
```

To do this refactoring with migration mechanics, I begin by applying *Extract Function (106)* to the entire function body.

```
function circum(radius) {
  return circumference(radius);
}
function circumference(radius) {
  return 2 * Math.PI * radius;
}
```

I test that, then apply *Inline Function (115)* to the old functions. I find all the calls of the old function and replace each one with a call of the new one. I can test after each change, which allows me to do them one at a time. Once I've got them all, I remove the old function.

With most refactorings, I'm changing code that I can modify, but this refactoring can be handy with a published API—that is, one used by code that I'm unable to change myself. I can pause the refactoring after creating `circumference` and, if possible, mark `circum` as deprecated. I will then wait for callers to change to use `circumference`; once they do, I can delete `circum`. Even if I'm never able to reach the happy point of deleting `circum`, at least I have a better name for new code.

Example: Adding a Parameter

In some software, to manage a library of books, I have a book class which has the ability to take a reservation for a customer.

class Book…

```
addReservation(customer) {
  this._reservations.push(customer);
}
```

I need to support a priority queue for reservations. Thus, I need an extra parameter on `addReservation` to indicate whether the reservation should go in the usual queue or the high-priority queue. If I can easily find and change all the callers, then I can just go ahead with the change—but if not, I can use the migration approach, which I'll show here.

I begin by using *Extract Function (106)* on the body of `addReservation` to create the new function. Although it will eventually be called `addReservation`, the new and old functions can't coexist with the same name. So I use a temporary name that will be easy to search for later.

class Book…

```
addReservation(customer) {
  this.zz_addReservation(customer);
}
```

```
zz_addReservation(customer) {
  this._reservations.push(customer);
}
```

I then add the parameter to the new declaration and its call (in effect, using the simple mechanics).

class Book…

```
addReservation(customer) {
  this.zz_addReservation(customer, false);
}

zz_addReservation(customer, isPriority) {
  this._reservations.push(customer);
}
```

When I use JavaScript, before I change any of the callers, I like to apply *Introduce Assertion (302)* to check the new parameter is used by the caller.

class Book…

```
zz_addReservation(customer, isPriority) {
  assert(isPriority === true || isPriority === false);
  this._reservations.push(customer);
}
```

Now, when I change the callers, if I make a mistake and leave off the new parameter, this assertion will help me catch the mistake. And I know from long experience there are few more mistake-prone programmers than myself.

Now, I can start changing the callers by using *Inline Function (115)* on the original function. This allows me to change one caller at a time.

I then rename the new function back to the original. Usually, the simple mechanics work fine for this, but I can also use the migration approach if I need to.

Example: Changing a Parameter to One of Its Properties

The examples so far are simple changes of a name and adding a new parameter, but with the migration mechanics, this refactoring can handle more complicated cases quite neatly. Here's an example that is a bit more involved.

I have a function which determines if a customer is based in New England.

```
function inNewEngland(aCustomer) {
  return ["MA", "CT", "ME", "VT", "NH", "RI"].includes(aCustomer.address.state);
}
```

Here is one of its callers:

caller…

```
const newEnglanders = someCustomers.filter(c => inNewEngland(c));
```

`inNewEngland` only uses the customer's home state to determine if it's in New England. I'd prefer to refactor `inNewEngland` so that it takes a state code as a parameter, making it usable in more contexts by removing the dependency on the customer.

With Change Function Declaration, my usual first move is to apply *Extract Function (106)*, but in this case I can make it easier by first refactoring the function body a little. I use *Extract Variable (119)* on my desired new parameter.

```
function inNewEngland(aCustomer) {
  const stateCode = aCustomer.address.state;
  return ["MA", "CT", "ME", "VT", "NH", "RI"].includes(stateCode);
}
```

Now I use *Extract Function (106)* to create that new function.

```
function inNewEngland(aCustomer) {
  const stateCode = aCustomer.address.state;
  return xxNEWinNewEngland(stateCode);
}

function xxNEWinNewEngland(stateCode) {
  return ["MA", "CT", "ME", "VT", "NH", "RI"].includes(stateCode);
}
```

I give the function a name that's easy to automatically replace to turn into the original name later. (You can tell I don't have a standard for these temporary names.)

I apply *Inline Variable (123)* on the input parameter in the original function.

```
function inNewEngland(aCustomer) {
  return xxNEWinNewEngland(aCustomer.address.state);
}
```

I use *Inline Function (115)* to fold the old function into its callers, effectively replacing the call to the old function with a call to the new one. I can do these one at a time.

caller…

```
const newEnglanders = someCustomers.filter(c => xxNEWinNewEngland(c.address.state));
```

Once I've inlined the old function into every caller, I use Change Function Declaration again to change the name of the new function to that of the original.

caller…

```
const newEnglanders = someCustomers.filter(c => inNewEngland(c.address.state));
```

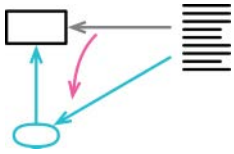
top level…

```
function inNewEngland(stateCode) {
  return ["MA", "CT", "ME", "VT", "NH", "RI"].includes(stateCode);
}
```

Automated refactoring tools make the migration mechanics both less useful and more effective. They make it less useful because they handle even complicated renames and parameter changes safer, so I don't have to use the migration approach as often as I do without that support. However, in cases like this example, where the tools can't do the whole refactoring, they still make it much easier as the key moves of extract and inline can be done more quickly and safely with the tool.

Encapsulate Variable

formerly: *Self-Encapsulate Field*
formerly: *Encapsulate Field*



```
let defaultOwner = {firstName: "Martin", lastName: "Fowler"};
```

```
let defaultOwnerData = {firstName: "Martin", lastName: "Fowler"};
export function defaultOwner()       {return defaultOwnerData;}
export function setDefaultOwner(arg) {defaultOwnerData = arg;}
```

Motivation

Refactoring is all about manipulating the elements of our programs. Data is more awkward to manipulate than functions. Since using a function usually means calling it, I can easily rename or move a function while keeping the old function intact as a forwarding function (so my old code calls the old function, which calls the new function). I'll usually not keep this forwarding function around for long, but it does simplify the refactoring.

Data is more awkward because I can't do that. If I move data around, I have to change all the references to the data in a single cycle to keep the code working. For data with a very small scope of access, such as a temporary variable in a small function, this isn't a problem. But as the scope grows, so does the difficulty, which is why global data is such a pain.

So if I want to move widely accessed data, often the best approach is to first encapsulate it by routing all its access through functions. That way, I turn the difficult task of reorganizing data into the simpler task of reorganizing functions.

Encapsulating data is valuable for other things too. It provides a clear point to monitor changes and use of the data; I can easily add validation or consequential logic on the updates. It is my habit to make all mutable data encapsulated like this and only accessed through functions if its scope is greater than a single function. The greater the scope of the data, the more important it is to encapsulate.

My approach with legacy code is that whenever I need to change or add a new reference to such a variable, I should take the opportunity to encapsulate it. That way I prevent the increase of coupling to commonly used data.

This principle is why the object-oriented approach puts so much emphasis on keeping an object's data private. Whenever I see a public field, I consider using Encapsulate Variable (in that case often called *Encapsulate Field*) to reduce its visibility. Some go further and argue that even internal references to fields within a class should go through accessor functions—an approach known as self-encapsulation. On the whole, I find self-encapsulation excessive—if a class is so big that I need to self-encapsulate its fields, it needs to be broken up anyway. But self-encapsulating a field is a useful step before splitting a class.

Keeping data encapsulated is much less important for immutable data. When the data doesn't change, I don't need a place to put in validation or other logic hooks before updates. I can also freely copy the data rather than move it—so I don't have to change references from old locations, nor do I worry about sections of code getting stale data. Immutability is a powerful preservative.

Mechanics

- Create encapsulating functions to access and update the variable.
- Run static checks.
- For each reference to the variable, replace with a call to the appropriate encapsulating function. Test after each replacement.
- Restrict the visibility of the variable.

  Sometimes it's not possible to prevent access to the variable. If so, it may be useful to detect any remaining references by renaming the variable and testing.
- Test.
- If the value of the variable is a record, consider *Encapsulate Record (162)*.

Example

Consider some useful data held in a global variable.

```
let defaultOwner = {firstName: "Martin", lastName: "Fowler"};
```

Like any data, it's referenced with code like this:

```
spaceship.owner = defaultOwner;
```

and updated like this:

```
defaultOwner = {firstName: "Rebecca", lastName: "Parsons"};
```

To do a basic encapsulation on this, I start by defining functions to read and write the data.

```
function getDefaultOwner()    {return defaultOwner;}
function setDefaultOwner(arg) {defaultOwner = arg;}
```

I then start working on references to `defaultOwner`. When I see a reference, I replace it with a call to the getting function.

```
spaceship.owner = getDefaultOwner();
```

When I see an assignment, I replace it with the setting function.

```
setDefaultOwner({firstName: "Rebecca", lastName: "Parsons"});
```

I test after each replacement.

Once I'm done with all the references, I restrict the visibility of the variable. This both checks that there aren't any references that I've missed, and ensures that future changes to the code won't access the variable directly. I can do that in JavaScript by moving both the variable and the accessor methods to their own file and only exporting the accessor methods.

defaultOwner.js…

```
let defaultOwner = {firstName: "Martin", lastName: "Fowler"};
export function getDefaultOwner()    {return defaultOwner;}
export function setDefaultOwner(arg) {defaultOwner = arg;}
```

If I'm in a situation where I cannot restrict the access to a variable, it may be useful to rename the variable and retest. That won't prevent future direct access, but naming the variable something meaningful and awkward such as `__privateOnly_defaultOwner` may help.

I don't like the use of `get` prefixes on getters, so I'll rename to remove it.

defaultOwner.js…

```
let defaultOwnerData = {firstName: "Martin", lastName: "Fowler"};
export function ~~get~~defaultOwner()       {return defaultOwnerData;}
export function setDefaultOwner(arg) {defaultOwnerData = arg;}
```

A common convention in JavaScript is to name a getting function and setting function the same and differentiate them due the presence of an argument. I call this practice Overloaded Getter Setter [mf-ogs] and strongly dislike it. So, even though I don't like the `get` prefix, I will keep the `set` prefix.

Encapsulating the Value

The basic refactoring I've outlined here encapsulates a reference to some data structure, allowing me to control its access and reassignment. But it doesn't control changes to that structure.

```
const owner1 = defaultOwner();
assert.equal("Fowler", owner1.lastName, "when set");
const owner2 = defaultOwner();
owner2.lastName = "Parsons";
assert.equal("Parsons", owner1.lastName, "after change owner2"); // is this ok?
```

The basic refactoring encapsulates the reference to the data item. In many cases, this is all I want to do for the moment. But I often want to take the encapsulation deeper to control not just changes to the variable but also to its contents.

For this, I have a couple of options. The simplest one is to prevent any changes to the value. My favorite way to handle this is by modifying the getting function to return a copy of the data.

defaultOwner.js…

```
let defaultOwnerData = {firstName: "Martin", lastName: "Fowler"};
export function defaultOwner()       {return Object.assign({}, defaultOwnerData);}
export function setDefaultOwner(arg) {defaultOwnerData = arg;}
```

I use this approach particularly often with lists. If I return a copy of the data, any clients using it can change it, but that change isn't reflected in the shared data. I have to be careful with using copies, however: Some code may expect to change shared data. If that's the case, I'm relying on my tests to detect a problem. An alternative is to prevent changes—and a good way of doing that is *Encapsulate Record (162)*.

```
let defaultOwnerData = {firstName: "Martin", lastName: "Fowler"};
export function defaultOwner()       {return new Person(defaultOwnerData);}
export function setDefaultOwner(arg) {defaultOwnerData = arg;}

class Person {
  constructor(data) {
    this._lastName = data.lastName;
    this._firstName = data.firstName
  }
  get lastName() {return this._lastName;}
  get firstName() {return this._firstName;}
  // and so on for other properties
```

Now, any attempt to reassign the properties of the default owner will cause an error. Different languages have different techniques to detect or prevent changes like this, so depending on the language I'd consider other options.

Detecting and preventing changes like this is often worthwhile as a temporary measure. I can either remove the changes, or provide suitable mutating functions. Then, once they are all dealt with, I can modify the getting method to return a copy.

So far I've talked about copying on getting data, but it may be worthwhile to make a copy in the setter too. That will depend on where the data comes from and whether I need to maintain a link to reflect any changes in that original data.

If I don't need such a link, a copy prevents accidents due to changes on that source data. Taking a copy may be superfluous most of the time, but copies in these cases usually have a negligible effect on performance; on the other hand, if I don't do them, there is a risk of a long and difficult bout of debugging in the future.

Remember that the copying above, and the class wrapper, both only work one level deep in the record structure. Going deeper requires more levels of copies or object wrapping.

As you can see, encapsulating data is valuable, but often not straightforward. Exactly what to encapsulate—and how to do it—depends on the way the data is being used and the changes I have in mind. But the more widely it's used, the more it's worth my attention to encapsulate properly.

Rename Variable


```
let a = height * width;
```

```
let area = height * width;
```

Motivation

Naming things well is the heart of clear programming. Variables can do a lot to explain what I'm up to—if I name them well. But I frequently get my names wrong—sometimes because I'm not thinking carefully enough, sometimes because my understanding of the problem improves as I learn more, and sometimes because the program's purpose changes as my users' needs change.

Even more than most program elements, the importance of a name depends on how widely it's used. A variable used in a one-line lambda expression is usually easy to follow—I often use a single letter in that case since the variable's purpose is clear from its context. Parameters for short functions can often be terse for the same reason, although in a dynamically typed language like JavaScript, I do like to put the type into the name (hence parameter names like `aCustomer`).

Persistent fields that last beyond a single function invocation require more careful naming. This is where I'm likely to put most of my attention.

Mechanics

- If the variable is used widely, consider *Encapsulate Variable (132)*.
- Find all references to the variable, and change every one.

  If there are references from another code base, the variable is a published variable, and you cannot do this refactoring.

  If the variable does not change, you can copy it to one with the new name, then change gradually, testing after each change.
- Test.

Example

The simplest case for renaming a variable is when it's local to a single function: a temp or argument. It's too trivial for even an example: I just find each reference and change it. After I'm done, I test to ensure I didn't mess up.

Problems occur when the variable has a wider scope than just a single function. There may be a lot of references all over the code base:

```
let tpHd = "untitled";
```

Some references access the variable:

```
result += `<h1>${tpHd}</h1>`;
```

Others update it:

```
tpHd = obj['articleTitle'];
```

My usual response to this is apply *Encapsulate Variable (132)*.

```
result += `<h1>${title()}</h1>`;

setTitle(obj['articleTitle']);

  function title()       {return tpHd;}
  function setTitle(arg) {tpHd = arg;}
```

At this point, I can rename the variable.

```
let _title = "untitled";

function title()       {return _title;}
function setTitle(arg) {_title = arg;}
```

I could continue by inlining the wrapping functions so all callers are using the variable directly. But I'd rarely want to do this. If the variable is used widely enough that I feel the need to encapsulate it in order to change its name, it's worth keeping it encapsulated behind functions for the future.

> In cases where I was going to inline, I'd call the getting function `getTitle` and not use an underscore for the variable name when I rename it.

Renaming a Constant

If I'm renaming a constant (or something that acts like a constant to clients) I can avoid encapsulation, and still do the rename gradually, by copying. If the original declaration looks like this:

```
const cpyNm = "Acme Gooseberries";
```

I can begin the renaming by making a copy:

```
const companyName = "Acme Gooseberries";
const cpyNm = companyName;
```

With the copy, I can gradually change references from the old name to the new name. When I'm done, I remove the copy. I prefer to declare the new name and copy to the old name if it makes it a tad easier to remove the old name and put it back again should a test fail.

This works for constants as well as for variables that are read-only to clients (such as an exported variable in JavaScript).

Introduce Parameter Object


```
function amountInvoiced(startDate, endDate) {...}
function amountReceived(startDate, endDate) {...}
function amountOverdue(startDate, endDate) {...}
```

```
function amountInvoiced(aDateRange) {...}
function amountReceived(aDateRange) {...}
function amountOverdue(aDateRange) {...}
```

Motivation

I often see groups of data items that regularly travel together, appearing in function after function. Such a group is a data clump, and I like to replace it with a single data structure.

Grouping data into a structure is valuable because it makes explicit the relationship between the data items. It reduces the size of parameter lists for any function that uses the new structure. It helps consistency since all functions that use the structure will use the same names to get at its elements.

But the real power of this refactoring is how it enables deeper changes to the code. When I identify these new structures, I can reorient the behavior of the program to use these structures. I will create functions that capture the common behavior over this data—either as a set of common functions or as a class that combines the data structure with these functions. This process can change the conceptual picture of the code, raising these structures as new abstractions that can greatly simplify my understanding of the domain. When this works, it can have surprisingly powerful effects—but none of this is possible unless I use Introduce Parameter Object to begin the process.

Mechanics

- If there isn't a suitable structure already, create one.

  I prefer to use a class, as that makes it easier to group behavior later on. I usually like to ensure these structures are value objects [mf-vo].

- Test.
- Use *Change Function Declaration (124)* to add a parameter for the new structure.
- Test.
- Adjust each caller to pass in the correct instance of the new structure. Test after each one.
- For each element of the new structure, replace the use of the original parameter with the element of the structure. Remove the parameter. Test.

Example

I'll begin with some code that looks at a set of temperature readings and determines whether any of them fall outside of an operating range. Here's what the data looks like for the readings:

```
const station = { name: "ZB1",
                  readings: [
                    {temp: 47, time: "2016-11-10 09:10"},
                    {temp: 53, time: "2016-11-10 09:20"},
                    {temp: 58, time: "2016-11-10 09:30"},
                    {temp: 53, time: "2016-11-10 09:40"},
                    {temp: 51, time: "2016-11-10 09:50"},
                  ]
                };
```

I have a function to find the readings that are outside a temperature range.

```
function readingsOutsideRange(station, min, max) {
  return station.readings
    .filter(r => r.temp < min || r.temp > max);
}
```

It might be called from some code like this:

caller

```
alerts = readingsOutsideRange(station,
                              operatingPlan.temperatureFloor,
                              operatingPlan.temperatureCeiling);
```

Notice how the calling code pulls the two data items as a pair from another object and passes the pair into `readingsOutsideRange`. The operating plan uses different names to indicate the start and end of the range compared to `readingsOutsideRange`. A range like this is a common case where two separate data items are better combined into a single object. I'll begin by declaring a class for the combined data.

```
class NumberRange {
  constructor(min, max) {
    this._data = {min: min, max: max};
  }
  get min() {return this._data.min;}
  get max() {return this._data.max;}
}
```

I declare a class, rather than just using a basic JavaScript object, because I usually find this refactoring to be a first step to moving behavior into the newly created object. Since a class makes sense for this, I go right ahead and use one directly. I also don't provide any update methods for the new class, as I'll probably make this a Value Object [mf-vo]. Most times I do this refactoring, I create value objects.

I then use *Change Function Declaration (124)* to add the new object as a parameter to `readingsOutsideRange`.

```
function readingsOutsideRange(station, min, max, range) {
  return station.readings
    .filter(r => r.temp < min || r.temp > max);
}
```

In JavaScript, I can leave the caller as is, but in other languages I'd have to add a null for the new parameter which would look something like this:

caller

```
alerts = readingsOutsideRange(station,
                              operatingPlan.temperatureFloor,
                              operatingPlan.temperatureCeiling,
                              null);
```

At this point I haven't changed any behavior, and tests should still pass. I then go to each caller and adjust it to pass in the correct date range.

caller

```
const range = new NumberRange(operatingPlan.temperatureFloor, operatingPlan.temperatureCeiling);
alerts = readingsOutsideRange(station,
                              operatingPlan.temperatureFloor,
                              operatingPlan.temperatureCeiling,
                              range);
```

I still haven't altered any behavior yet, as the parameter isn't used. All tests should still work.

Now I can start replacing the usage of the parameters. I'll start with the maximum.

```
function readingsOutsideRange(station, min, max, range) {
  return station.readings
    .filter(r => r.temp < min || r.temp > range.max);
}
```

caller

```
const range = new NumberRange(operatingPlan.temperatureFloor, operatingPlan.temperatureCeiling);
alerts = readingsOutsideRange(station,
                              operatingPlan.temperatureFloor,
                              operatingPlan.temperatureCeiling,
                              range);
```

I can test at this point, then remove the other parameter.

```
function readingsOutsideRange(station, min, range) {
  return station.readings
    .filter(r => r.temp < range.min || r.temp > range.max);
}
```

caller

```
const range = new NumberRange(operatingPlan.temperatureFloor, operatingPlan.temperatureCeiling);
alerts = readingsOutsideRange(station,
                              operatingPlan.temperatureFloor,
                              range);
```

That completes this refactoring. However, replacing a clump of parameters with a real object is just the setup for the really good stuff. The great benefits of making a class like this is that I can then move behavior into the new class. In this case, I'd add a method for range that tests if a value falls within the range.

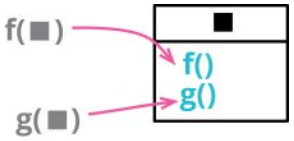
```
function readingsOutsideRange(station, range) {
  return station.readings
    .filter(r => !range.contains(r.temp));
}
```

class NumberRange…

```
  contains(arg) {return (arg >= this.min && arg <= this.max);}
```

This is a first step to creating a range [mf-range] that can take on a lot of useful behavior. Once I've identified the need for a range in my code, I can be constantly on the lookout for other cases where I see a max/min pair of numbers and replace them with a range. (One immediate possibility is the operating plan, replacing `temperatureFloor` and `temperatureCeiling` with a `temperatureRange`.) As I look at how these pairs are used, I can move more useful behavior into the range class, simplifying its usage across the code base. One of the first things I may add is a value-based equality method to make it a true value object.

Combine Functions into Class




```
function base(aReading) {...}
function taxableCharge(aReading) {...}
function calculateBaseCharge(aReading) {...}
```

```
class Reading {
  base() {...}
  taxableCharge() {...}
  calculateBaseCharge() {...}
}
```

Motivation

Classes are a fundamental construct in most modern programming languages. They bind together data and functions into a shared environment, exposing some of that data and function to other program elements for collaboration. They are the primary construct in object-oriented languages, but are also useful with other approaches too.

When I see a group of functions that operate closely together on a common body of data (usually passed as arguments to the function call), I see an opportunity to form a class. Using a class makes the common environment that these functions share more explicit, allows me to simplify function calls inside the object by removing many of the arguments, and provides a reference to pass such an object to other parts of the system.

In addition to organizing already formed functions, this refactoring also provides a good opportunity to identify other bits of computation and refactor them into methods on the new class.

Another way of organizing functions together is *Combine Functions into Transform (149)*. Which one to use depends more on the broader context of the program. One significant advantage of using a class is that it allows clients to mutate the core data of the object, and the derivations remain consistent.

As well as a class, functions like this can also be combined into a nested function. Usually I prefer a class to a nested function, as it can be difficult to test functions nested within another. Classes are also necessary when there is more than one function in the group that I want to expose to collaborators.

Languages that don't have classes as a first-class element, but do have first-class functions, often use the Function As Object [mf-fao] to provide this capability.

Mechanics

- Apply *Encapsulate Record (162)* to the common data record that the functions share.

  If the data that is common between the functions isn't already grouped into a record structure, use *Introduce Parameter Object (140)* to create a record to group it together.

- Take each function that uses the common record and use *Move Function (198)* to move it into the new class.

  Any arguments to the function call that are members can be removed from the argument list.

- Each bit of logic that manipulates the data can be extracted with *Extract Function (106)* and then moved into the new class.

Example

I grew up in England, a country renowned for its love of Tea. (Personally, I don't like most tea they serve in England, but have since acquired a taste for Chinese and Japanese teas.) So my author's fantasy conjures up a state utility for providing tea to the population. Every month they read the tea meters, to get a record like this:

```
reading = {customer: "ivan", quantity: 10, month: 5, year: 2017};
```

I look through the code that processes these records, and I see lots of places where similar calculations are done on the data. So I find a spot that calculates the base charge:

client 1…

```
const aReading = acquireReading();
const baseCharge = baseRate(aReading.month, aReading.year) * aReading.quantity;
```

Being England, everything essential must be taxed, so it is with tea. But the rules allow at least an essential level of tea to be free of taxation.

client 2…

```
const aReading = acquireReading();
const base = (baseRate(aReading.month, aReading.year) * aReading.quantity);
const taxableCharge =  Math.max(0, base - taxThreshold(aReading.year));
```

I'm sure that, like me, you noticed that the formula for the base charge is duplicated between these two fragments. If you're like me, you're already reaching for *Extract Function (106)*. Interestingly, it seems our work has been done for us elsewhere.

client 3…

```
const aReading = acquireReading();
const basicChargeAmount = calculateBaseCharge(aReading);

function calculateBaseCharge(aReading) {
  return  baseRate(aReading.month, aReading.year) * aReading.quantity;
}
```

Given this, I have a natural impulse to change the two earlier bits of client code to use this function. But the trouble with top-level functions like this is that they are often easy to miss. I'd rather change the code to give the function a closer connection to the data it processes. A good way to do this is to turn the data into a class.

To turn the record into a class, I use *Encapsulate Record (162)*.

```
class Reading {
  constructor(data) {
    this._customer = data.customer;
    this._quantity = data.quantity;
    this._month = data.month;
    this._year = data.year;
  }
  get customer() {return this._customer;}
  get quantity() {return this._quantity;}
  get month()    {return this._month;}
  get year()     {return this._year;}
}
```

To move the behavior, I'll start with the function I already have: `calculateBaseCharge`. To use the new class, I need to apply it to the data as soon as I've acquired it.

client 3…

```
const rawReading = acquireReading();
const aReading = new Reading(rawReading);
const basicChargeAmount = calculateBaseCharge(aReading);
```

I then use *Move Function (198)* to move `calculateBaseCharge` into the new class.

class Reading…

```
get calculateBaseCharge() {
  return  baseRate(this.month, this.year) * this.quantity;
}
```

client 3…

```
const rawReading = acquireReading();
const aReading = new Reading(rawReading);
const basicChargeAmount = aReading.calculateBaseCharge;
```

While I'm at it, I use *Rename Function (124)* to make it something more to my liking.

```
get baseCharge() {
  return  baseRate(this.month, this.year) * this.quantity;
}
```

client 3…

```
const rawReading = acquireReading();
const aReading = new Reading(rawReading);
const basicChargeAmount = aReading.baseCharge;
```

With this naming, the client of the reading class can't tell whether the base charge is a field or a derived value. This is a Good Thing—the Uniform Access Principle [mf-ua].

I now alter the first client to call the method rather than repeat the calculation.

client 1…

```
const rawReading = acquireReading();
const aReading = new Reading(rawReading);
const baseCharge = aReading.baseCharge;
```

There's a strong chance I'll use *Inline Variable (123)* on the `baseCharge` variable before the day is out. But more relevant to this refactoring is the client that calculates the taxable amount. My first step here is to use the new base charge property.

client 2…

```
const rawReading = acquireReading();
const aReading = new Reading(rawReading);
const taxableCharge =  Math.max(0, aReading.baseCharge - taxThreshold(aReading.year));
```

I use *Extract Function (106)* on the calculation for the taxable charge.

```
function taxableChargeFn(aReading) {
  return  Math.max(0, aReading.baseCharge - taxThreshold(aReading.year));
}
```

client 3…

```
const rawReading = acquireReading();
const aReading = new Reading(rawReading);
const taxableCharge = taxableChargeFn(aReading);
```

Then I apply *Move Function (198)*.

class Reading…

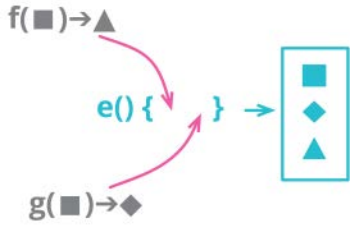
```
  get taxableCharge() {
    return  Math.max(0, this.baseCharge - taxThreshold(this.year));
  }
```

client 3…

```
const rawReading = acquireReading();
const aReading = new Reading(rawReading);
const taxableCharge = aReading.taxableCharge;
```

Since all the derived data is calculated on demand, I have no problem should I need to update the stored data. In general, I prefer immutable data, but many circumstances force us to work with mutable data (such as JavaScript, a language ecosystem that wasn't designed with immutability in mind). When there is a reasonable chance the data will be updated somewhere in the program, then a class is very helpful.

Combine Functions into Transform




```
function base(aReading) {...}
function taxableCharge(aReading) {...}
```

```
function enrichReading(argReading) {
  const aReading = _.cloneDeep(argReading);
  aReading.baseCharge = base(aReading);
  aReading.taxableCharge = taxableCharge(aReading);
  return aReading;
}
```

Motivation

Software often involves feeding data into programs that calculate various derived information from it. These derived values may be needed in several places, and those calculations are often repeated wherever the derived data is used. I prefer to bring all of these derivations together, so I have a consistent place to find and update them and avoid any duplicate logic.

One way to do this is to use a data transformation function that takes the source data as input and calculates all the derivations, putting each derived value as a field in the output data. Then, to examine the derivations, all I need do is look at the transform function.

An alternative to Combine Functions into Transform is *Combine Functions into Class (144)* that moves the logic into methods on a class formed from the source data. Either of these refactorings are helpful, and my choice will often depend on the style of programming already in the software. But there is one important difference: Using a class is much better if the source data gets updated within the code. Using a transform stores derived data in the new record, so if the source data changes, I will run into inconsistencies.

One of the reasons I like to do combine functions is to avoid duplication of the derivation logic. I can do that just by using *Extract Function (106)* on the logic, but it's often difficult to find the functions unless they are kept close to the data structures they operate on. Using a transform (or a class) makes it easy to find and use them.

Mechanics

- Create a transformation function that takes the record to be transformed and returns the same values.

  This will usually involve a deep copy of the record. It is often worthwhile to write a test to ensure the transform does not alter the original record.

- Pick some logic and move its body into the transform to create a new field in the record. Change the client code to access the new field.

  If the logic is complex, use *Extract Function (106)* first.

- Test.

- Repeat for the other relevant functions.

Example

Where I grew up, tea is an important part of life—so much that I can imagine a special utility that provides tea to the populace that's regulated like a utility. Every month, the utility gets a reading of how much tea a customer has acquired.

```
reading = {customer: "ivan", quantity: 10, month: 5, year: 2017};
```

Code in various places calculates various consequences of this tea usage. One such calculation is the base monetary amount that's used to calculate the charge for the customer.

client 1…

```
const aReading = acquireReading();
const baseCharge = baseRate(aReading.month, aReading.year) * aReading.quantity;
```

Another is the amount that should be taxed—which is less than the base amount since the government wisely considers that every citizen should get some tea tax free.

client 2…

```
const aReading = acquireReading();
const base = (baseRate(aReading.month, aReading.year) * aReading.quantity);
const taxableCharge =  Math.max(0, base - taxThreshold(aReading.year));
```

Looking through this code, I see these calculations repeated in several places. Such duplication is asking for trouble when they need to change (and I'd bet it's "when" not "if"). I can deal with this repetition by using *Extract Function (106)* on these calculations, but such functions often end up scattered around the program making it hard for future developers to realize they are there. Indeed, looking around I discover such a function, used in another area of the code.

client 3…

```
const aReading = acquireReading();
const basicChargeAmount = calculateBaseCharge(aReading);

function calculateBaseCharge(aReading) {
  return  baseRate(aReading.month, aReading.year) * aReading.quantity;
}
```

One way of dealing with this is to move all of these derivations into a transformation step that takes the raw reading and emits a reading enriched with all the common derived results.

I begin by creating a transformation function that merely copies the input object.

```
function enrichReading(original) {
  const result = _.cloneDeep(original);
  return result;
}
```

I'm using the `cloneDeep` *from lodash to create a deep copy.*

When I'm applying a transformation that produces essentially the same thing but with additional information, I like to name it using "enrich". If it were producing something I felt was different, I would name it using "transform".

I then pick one of the calculations I want to change. First, I enrich the reading it uses with the current one that does nothing yet.

client 3…

```
const rawReading = acquireReading();
const aReading = enrichReading(rawReading);
const basicChargeAmount = calculateBaseCharge(aReading);
```

I use *Move Function (198)* on `calculateBaseCharge` to move it into the enrichment calculation.

```
function enrichReading(original) {
  const result = _.cloneDeep(original);
  result.baseCharge = calculateBaseCharge(result);
  return result;
}
```

Within the transformation function, I'm happy to mutate a result object, instead of copying each time. I like immutability, but most common languages make it difficult to work with. I'm prepared to go through the extra effort to support

it at boundaries, but will mutate within smaller scopes. I also pick my names (using `aReading` as the accumulating variable) to make it easier to move the code into the transformer function.

I change the client that uses that function to use the enriched field instead.

client 3…

```
const rawReading = acquireReading();
const aReading = enrichReading(rawReading);
const basicChargeAmount = aReading.baseCharge;
```

Once I've moved all calls to `calculateBaseCharge`, I can nest it inside `enrichReading`. That would make it clear that clients that need the calculated base charge should use the enriched record.

One trap to beware of here. When I write `enrichReading` like this, to return the enriched reading, I'm implying that the original reading record isn't changed. So it's wise for me to add a test.

```
it('check reading unchanged', function() {
  const baseReading = {customer: "ivan", quantity: 15, month: 5, year: 2017};
  const oracle = _.cloneDeep(baseReading);
  enrichReading(baseReading);
  assert.deepEqual(baseReading, oracle);
});
```

I can then change client 1 to also use the same field.

client 1…

```
const rawReading = acquireReading();
const aReading = enrichReading(rawReading);
const baseCharge = aReading.baseCharge;
```

There is a good chance I can then use *Inline Variable (123)* on `baseCharge` too.

Now I turn to the taxable amount calculation. My first step is to add in the transformation function.

```
const rawReading = acquireReading();
const aReading = enrichReading(rawReading);
const base = (baseRate(aReading.month, aReading.year) * aReading.quantity);
const taxableCharge =  Math.max(0, base - taxThreshold(aReading.year));
```

I can immediately replace the calculation of the base charge with the new field. If the calculation was complex, I could *Extract Function (106)* first, but here it's simple enough to do in one step.

```
const rawReading = acquireReading();
const aReading = enrichReading(rawReading);
const base = aReading.baseCharge;
const taxableCharge =  Math.max(0, base - taxThreshold(aReading.year));
```

Once I've tested that that works, I apply *Inline Variable (123)*:

```
const rawReading = acquireReading();
const aReading = enrichReading(rawReading);
const taxableCharge =  Math.max(0, aReading.baseCharge - taxThreshold(aReading.year));
```

and move that computation into the transformer:

```
function enrichReading(original) {
  const result = _.cloneDeep(original);
  result.baseCharge = calculateBaseCharge(result);
  result.taxableCharge = Math.max(0, result.baseCharge - taxThreshold(result.year));
  return result;
}
```

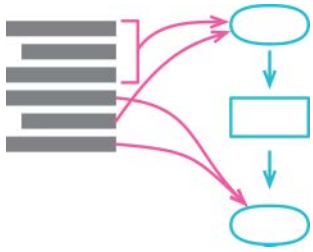
I modify the original code to use the new field.

```
const rawReading = acquireReading();
const aReading = enrichReading(rawReading);
const taxableCharge = aReading.taxableCharge;
```

Once I've tested that, it's likely I would be able to use *Inline Variable (123)* on `taxableCharge`.

One big problem with an enriched reading like this is: What happens should a client change a data value? Changing, say, the quantity field would result in data that's inconsistent. To avoid this in JavaScript, my best option is to use *Combine Functions into Class (144)* instead. If I'm in a language with immutable data structures, I don't have this problem, so its more common to see transforms in those languages. But even in languages without immutability, I can use transforms if the data appears in a read-only context, such as deriving data to display on a web page.

Split Phase



```
const orderData = orderString.split(/\s+/);
const productPrice = priceList[orderData[0].split("-")[1]];
const orderPrice = parseInt(orderData[1]) * productPrice;
```

```
const orderRecord = parseOrder(order);
const orderPrice = price(orderRecord, priceList);

function parseOrder(aString) {
  const values =  aString.split(/\s+/);
  return ({
    productID: values[0].split("-")[1],
    quantity: parseInt(values[1]),
  });
}
function price(order, priceList) {
  return order.quantity * priceList[order.productID];
}
```

Motivation

When I run into code that's dealing with two different things, I look for a way to split it into separate modules. I endeavor to make this split because, if I need to make a change, I can deal with each topic separately and not have to hold both in my head together. If I'm lucky, I may only have to change one module without having to remember the details of the other one at all.

One of the neatest ways to do a split like this is to divide the behavior into two sequential phases. A good example of this is when you have some processing whose inputs don't reflect the model you need to carry out the logic. Before you begin, you can massage the input into a convenient form for your main processing.

Or, you can take the logic you need to do and break it down into sequential steps, where each step is significantly different in what it does.

The most obvious example of this is a compiler. It's a basic task is to take some text (code in a programming language) and turn it into some executable form (e.g., object code for a specific hardware). Over time, we've found this can be usefully split into a chain of phases: tokenizing the text, parsing the tokens into a syntax tree, then various steps of transforming the syntax tree (e.g., for optimization), and finally generating the object code. Each step has a limited scope and I can think of one step without understanding the details of others.

Splitting phases like this is common in large software; the various phases in a compiler can each contain many functions and classes. But I can carry out the basic split-phase refactoring on any fragment of code—whenever I see an opportunity to usefully separate the code into different phases. The best clue is when different stages of the fragment use different sets of data and functions. By turning them into separate modules I can make this difference explicit, revealing the difference in the code.

Mechanics

- Extract the second phase code into its own function.
- Test.
- Introduce an intermediate data structure as an additional argument to the extracted function.
- Test.
- Examine each parameter of the extracted second phase. If it is used by first phase, move it to the intermediate data structure. Test after each move.

  Sometimes, a parameter should not be used by the second phase. In this case, extract the results of each usage of the parameter into a field of the intermediate data structure and use *Move Statements to Callers (217)* on the line that populates it.
- Apply *Extract Function (106)* on the first-phase code, returning the intermediate data structure.

  It's also reasonable to extract the first phase into a transformer object.

Example

I'll start with code to price an order for some vague and unimportant kind of goods:

```
function priceOrder(product, quantity, shippingMethod) {
  const basePrice = product.basePrice * quantity;
  const discount = Math.max(quantity - product.discountThreshold, 0)
          * product.basePrice * product.discountRate;
  const shippingPerCase = (basePrice > shippingMethod.discountThreshold)
          ? shippingMethod.discountedFee : shippingMethod.feePerCase;
  const shippingCost = quantity * shippingPerCase;
  const price =  basePrice - discount + shippingCost;
  return price;
}
```

Although this is the usual kind of trivial example, there is a sense of two phases going on here. The first couple of lines of code use the product information to calculate the product-oriented price of the order, while the later code uses shipping information to determine the shipping cost. If I have changes coming up that complicate the pricing and shipping calculations, but they work relatively independently, then splitting this code into two phases is valuable.

I begin by applying *Extract Function (106)* to the shipping calculation.

```
function priceOrder(product, quantity, shippingMethod) {
  const basePrice = product.basePrice * quantity;
  const discount = Math.max(quantity - product.discountThreshold, 0)
          * product.basePrice * product.discountRate;
  const price =  applyShipping(basePrice, shippingMethod, quantity, discount);
  return price;
}
function applyShipping(basePrice, shippingMethod, quantity, discount) {
  const shippingPerCase = (basePrice > shippingMethod.discountThreshold)
          ? shippingMethod.discountedFee : shippingMethod.feePerCase;
  const shippingCost = quantity * shippingPerCase;
  const price =  basePrice - discount + shippingCost;
  return price;
}
```

I pass in all the data that this second phase needs as individual parameters. In a more realistic case, there can be a lot of these, but I don't worry about it as I'll whittle them down later.

Next, I introduce the intermediate data structure that will communicate between the two phases.

```
function priceOrder(product, quantity, shippingMethod) {
  const basePrice = product.basePrice * quantity;
  const discount = Math.max(quantity - product.discountThreshold, 0)
          * product.basePrice * product.discountRate;
  const priceData = {};
  const price =  applyShipping(priceData, basePrice, shippingMethod, quantity, discount);
  return price;
}
```

```
function applyShipping(priceData, basePrice, shippingMethod, quantity, discount) {
  const shippingPerCase = (basePrice > shippingMethod.discountThreshold)
          ? shippingMethod.discountedFee : shippingMethod.feePerCase;
  const shippingCost = quantity * shippingPerCase;
  const price =  basePrice - discount + shippingCost;
  return price;
}
```

Now, I look at the various parameters to `applyShipping`. The first one is `basePrice` which is created by the first-phase code. So I move this into the intermediate data structure, removing it from the parameter list.

```
function priceOrder(product, quantity, shippingMethod) {
  const basePrice = product.basePrice * quantity;
  const discount = Math.max(quantity - product.discountThreshold, 0)
          * product.basePrice * product.discountRate;
  const priceData = {basePrice: basePrice};
  const price =  applyShipping(priceData, basePrice, shippingMethod, quantity, discount);
  return price;
}
function applyShipping(priceData, basePrice, shippingMethod, quantity, discount) {
  const shippingPerCase = (priceData.basePrice > shippingMethod.discountThreshold)
          ? shippingMethod.discountedFee : shippingMethod.feePerCase;
  const shippingCost = quantity * shippingPerCase;
  const price =  priceData.basePrice - discount + shippingCost;
  return price;
}
```

The next parameter in the list is `shippingMethod`. This one I leave as is, since it isn't used by the first-phase code.

After this, I have `quantity`. This is used by the first phase but not created by it, so I could actually leave this in the parameter list. My usual preference, however, is to move as much as I can to the intermediate data structure.

```
function priceOrder(product, quantity, shippingMethod) {
  const basePrice = product.basePrice * quantity;
  const discount = Math.max(quantity - product.discountThreshold, 0)
          * product.basePrice * product.discountRate;
  const priceData = {basePrice: basePrice, quantity: quantity};
  const price =  applyShipping(priceData, shippingMethod, quantity, discount);
  return price;
}
function applyShipping(priceData, shippingMethod, quantity, discount) {
  const shippingPerCase = (priceData.basePrice > shippingMethod.discountThreshold)
          ? shippingMethod.discountedFee : shippingMethod.feePerCase;
  const shippingCost = priceData.quantity * shippingPerCase;
  const price =  priceData.basePrice - discount + shippingCost;
  return price;
}
```

I do the same with `discount`.

```
function priceOrder(product, quantity, shippingMethod) {
  const basePrice = product.basePrice * quantity;
  const discount = Math.max(quantity - product.discountThreshold, 0)
          * product.basePrice * product.discountRate;
  const priceData = {basePrice: basePrice, quantity: quantity, discount:discount};
  const price =  applyShipping(priceData, shippingMethod, discount);
  return price;
}
function applyShipping(priceData, shippingMethod, discount) {
  const shippingPerCase = (priceData.basePrice > shippingMethod.discountThreshold)
          ? shippingMethod.discountedFee : shippingMethod.feePerCase;
  const shippingCost = priceData.quantity * shippingPerCase;
  const price =  priceData.basePrice - priceData.discount + shippingCost;
  return price;
}
```

Once I've gone through all the function parameters, I have the intermediate data structure fully formed. So I can extract the first-phase code into its own function, returning this data.

```
function priceOrder(product, quantity, shippingMethod) {
  const priceData = calculatePricingData(product, quantity);
  const price =  applyShipping(priceData, shippingMethod);
  return price;
}
function calculatePricingData(product, quantity) {
  const basePrice = product.basePrice * quantity;
  const discount = Math.max(quantity - product.discountThreshold, 0)
          * product.basePrice * product.discountRate;
  return {basePrice: basePrice, quantity: quantity, discount:discount};
}
function applyShipping(priceData, shippingMethod) {
  const shippingPerCase = (priceData.basePrice > shippingMethod.discountThreshold)
          ? shippingMethod.discountedFee : shippingMethod.feePerCase;
  const shippingCost = priceData.quantity * shippingPerCase;
  const price =  priceData.basePrice - priceData.discount + shippingCost;
  return price;
}
```

I can't resist tidying out those final constants.

```
function priceOrder(product, quantity, shippingMethod) {
  const priceData = calculatePricingData(product, quantity);
  return applyShipping(priceData, shippingMethod);
}
```

```
function calculatePricingData(product, quantity) {
  const basePrice = product.basePrice * quantity;
  const discount = Math.max(quantity - product.discountThreshold, 0)
          * product.basePrice * product.discountRate;
  return {basePrice: basePrice, quantity: quantity, discount:discount};
}
function applyShipping(priceData, shippingMethod) {
  const shippingPerCase = (priceData.basePrice > shippingMethod.discountThreshold)
          ? shippingMethod.discountedFee : shippingMethod.feePerCase;
  const shippingCost = priceData.quantity * shippingPerCase;
  return priceData.basePrice - priceData.discount + shippingCost;
}
```