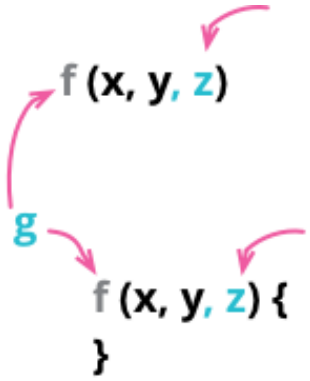
Change Function Declaration

previous: Inline Variable

next: Encapsulate Variable




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

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

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

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 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 to the old function.
- If you used a temporary name, use Change Function Declaration 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 declaration 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 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 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 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 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 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, but in this case I can make it easier by first refactoring the function body a little. I use Extract

Variable 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 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 on the input parameter in the original function.

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

I use Inline Function 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.

previous:
Inline Variable

next:
Encapsulate Variable