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.

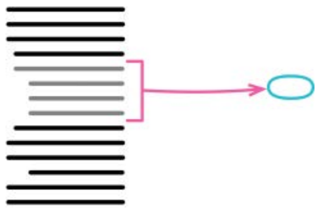
Probably the most common refactoring I do is extracting code into a function (*Extract Function (106)*) or a variable (*Extract Variable (119)*). Since refactoring is all about change, it's no surprise that I also frequently use the inverses of those two (*Inline Function (115)* and *Inline Variable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

$\Downarrow$

```
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`*.*

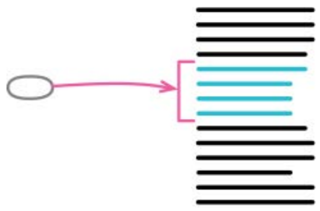
At this point you may be wondering, "What happens if more than one variable needs to be returned?"

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.

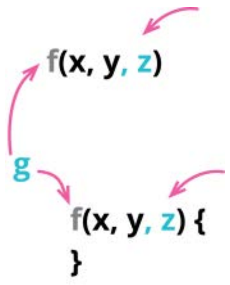
### Mechanics

- 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 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 (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));
```

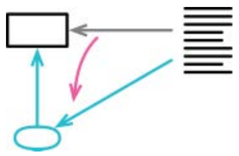
*top level…*

```
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"};
```

$\Downarrow$

```
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.

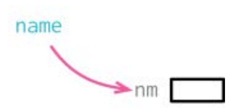
So far I've talked about copying on getting data, but it may be worthwhile to make a copy in the setter too. That will depend on where the data comes from and whether I need to maintain a link to reflect any changes in that original data.

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.

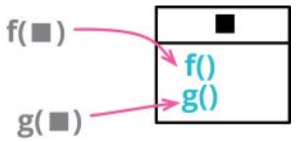
```
function readingsOutsideRange(station, range) {
  return station.readings
    .filter(r => !range.contains(r.temp));
}
```

*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…*

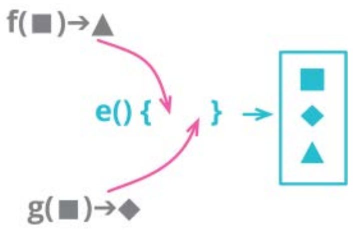
```
  get taxableCharge() {
    return  Math.max(0, this.baseCharge - taxThreshold(this.year));
  }
```

*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;
}
```

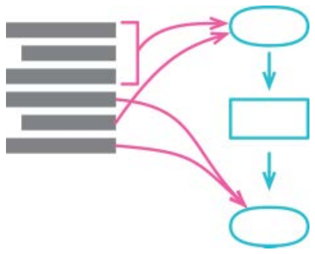
I modify the original code to use the new field.

```
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;
}
```

*This page intentionally left blank*

Chapter 7

# Encapsulation

Perhaps the most important criteria to be used in decomposing modules is to identify secrets that modules should hide from the rest of the system [Parnas]. Data structures are the most common secrets, and I can hide data structures by encapsulating them with *Encapsulate Record (162)* and *Encapsulate Collection (170)*. Even primitive data values can be encapsulated with *Replace Primitive with Object (174)*—the magnitude of second-order benefits from doing this often surprises people. Temporary variables often get in the way of refactoring—I have to ensure they are calculated in the right order and their values are available to other parts of the code that need them. Using *Replace Temp with Query (178)* is a great help here, particularly when splitting up an overly long function.

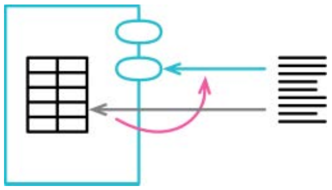
Classes were designed for information hiding. In the previous chapter, I described a way to form them with *Combine Functions into Class (144)*. The common extract/inline operations also apply to classes with *Extract Class (182)* and *Inline Class (186)*.

As well as hiding the internals of classes, it's often useful to hide connections between classes, which I can do with *Hide Delegate (189)*. But too much hiding leads to bloated interfaces, so I also need its reverse: *Remove Middle Man (192)*.

Classes and modules are the largest forms of encapsulation, but functions also encapsulate their implementation. Sometimes, I may need to make a wholesale change to an algorithm, which I can do by wrapping it in a function with *Extract Function (106)* and applying *Substitute Algorithm (195)*.

## Encapsulate Record

formerly: *Replace Record with Data Class*



```
organization = {name: "Acme Gooseberries", country: "GB"};
```

⇓

```
class Organization {
  constructor(data) {
    this._name = data.name;
    this._country = data.country;
  }
  get name()    {return this._name;}
  set name(arg) {this._name = arg;}
  get country()    {return this._country;}
  set country(arg) {this._country = arg;}
}
```

### Motivation

This is why I often favor objects over records for mutable data. With objects, I can hide what is stored and provide methods for all three values. The user of the object doesn't need to know or care which is stored and which is calculated. This encapsulation also helps with renaming: I can rename the field while providing methods for both the new and the old names, gradually updating callers until they are all done.

I just said I favor objects for *mutable* data. If I have an immutable value, I can just have all three values in my record, using an enrichment step if necessary. Similarly, it's easy to copy the field when renaming.

I can have two kinds of record structures: those where I declare the legal field names and those that allow me to use whatever I like. The latter are often implemented through a library class called something like hash, map, hashmap, dictionary, or associative array. Many languages provide convenient syntax for creating hashmaps, which makes them useful in many programming situations. The downside of using them is they are aren't explicit about their fields. The only way I can tell if they use start/end or start/length is by looking at where they are created and used. This isn't a problem if they are only used in a small section of a program, but the wider their scope of usage, the greater problem I get from their implicit structure. I could refactor such implicit records into explicit ones—but if I need to do that, I'd rather make them classes instead.

It's common to pass nested structures of lists and hashmaps which are often serialized into formats like JSON or XML. Such structures can be encapsulated too, which helps if their formats change later on or if I'm concerned about updates to the data that are hard to keep track of.

## *Mechanics*

- Use *Encapsulate Variable (132)* on the variable holding the record.

  Give the functions that encapsulate the record names that are easily searchable.

- Replace the content of the variable with a simple class that wraps the record. Define an accessor inside this class that returns the raw record. Modify the functions that encapsulate the variable to use this accessor.

- Test.

- Provide new functions that return the object rather than the raw record.

- For each user of the record, replace its use of a function that returns the record with a function that returns the object. Use an accessor on the object to get at the field data, creating that accessor if needed. Test after each change.

  If it's a complex record, such as one with a nested structure, focus on clients that update the data first. Consider returning a copy or read-only proxy of the data for clients that only read the data.

- Remove the class's raw data accessor and the easily searchable functions that returned the raw record.

- Test.

- If the fields of the record are themselves structures, consider using Encapsulate Record and *Encapsulate Collection (170)* recursively.

## *Example*

I'll start with a constant that is widely used across a program.

```
const organization = {name: "Acme Gooseberries", country: "GB"};
```

This is a JavaScript object which is being used as a record structure by various parts of the program, with accesses like this:

```
result += `<h1>${organization.name}</h1>`;
```

and

```
organization.name = newName;
```

The first step is a simple *Encapsulate Variable (132)*.

```
function getRawDataOfOrganization() {return organization;}
```

*example reader…*

```
result += `<h1>${getRawDataOfOrganization().name}</h1>`;
```

*example writer…*

```
getRawDataOfOrganization().name = newName;
```

It's not quite a standard *Encapsulate Variable (132)*, since I gave the getter a name deliberately chosen to be both ugly and easy to search for. This is because I intend its life to be short.

Encapsulating a record means going deeper than just the variable itself; I want to control how it's manipulated. I can do this by replacing the record with a class.

*class Organization…*

```
class Organization {
  constructor(data) {
    this._data = data;
  }
}
```

*top level*

```
const organization = new Organization({name: "Acme Gooseberries", country: "GB"});

function getRawDataOfOrganization() {return organization._data;}
function getOrganization() {return organization;}
```

Now that I have an object in place, I start looking at the users of the record. Any one that updates the record gets replaced with a setter.

*class Organization…*

```
set name(aString) {this._data.name = aString;}
```

*client…*

```
getOrganization().name = newName;
```

Similarly, I replace any readers with the appropriate getter.

*class Organization…*

```
get name()    {return this._data.name;}
```

*client…*

```
result += `<h1>${getOrganization().name}</h1>`;
```

After I've done that, I can follow through on my threat to give the ugly sounding function a short life.

```
~~function getRawDataOfOrganization() {return organization._data;}~~
function getOrganization() {return organization;}
```

I'd also be inclined to fold the `_data` field directly into the object.

```
class Organization {
  constructor(data) {
    this._name = data.name;
    this._country = data.country;
  }
  get name()    {return this._name;}
  set name(aString) {this._name = aString;}
  get country()    {return this._country;}
  set country(aCountryCode) {this._country = aCountryCode;}
}
```

This has the advantage of breaking the link to the input data record. This might be useful if a reference to it runs around, which would break encapsulation. Should I not fold the data into individual fields, I would be wise to copy `_data` when I assign it.

### *Example: Encapsulating a Nested Record*

The above example looks at a shallow record, but what do I do with data that is deeply nested, e.g., coming from a JSON document? The core refactoring steps still apply, and I have to be equally careful with updates, but I do get some options around reads.

As an example, here is some slightly more nested data: a collection of customers, kept in a hashmap indexed by their customer ID.

```
"1920": {
  name: "martin",
  id: "1920",
  usages: {
    "2016": {
      "1": 50,
      "2": 55,
      // remaining months of the year
    },
    "2015": {
      "1": 70,
      "2": 63,
      // remaining months of the year
    }
  }
},
"38673": {
  name: "neal",
  id: "38673",
  // more customers in a similar form
```

With more nested data, reads and writes can be digging into the data structure.

*sample update…*

```
customerData[customerID].usages[year][month] = amount;
```

*sample read…*

```
function compareUsage (customerID, laterYear, month) {
  const later   = customerData[customerID].usages[laterYear][month];
  const earlier = customerData[customerID].usages[laterYear - 1][month];
  return {laterAmount: later, change: later - earlier};
}
```

To encapsulate this data, I also start with *Encapsulate Variable (132)*.

```
function getRawDataOfCustomers()    {return customerData;}
function setRawDataOfCustomers(arg) {customerData = arg;}
```

*sample update…*

```
getRawDataOfCustomers()[customerID].usages[year][month] = amount;
```

*sample read…*

```
function compareUsage (customerID, laterYear, month) {
  const later   = getRawDataOfCustomers()[customerID].usages[laterYear][month];
  const earlier = getRawDataOfCustomers()[customerID].usages[laterYear - 1][month];
  return {laterAmount: later, change: later - earlier};
}
```

I then make a class for the overall data structure.

```
class CustomerData {
  constructor(data) {
    this._data = data;
  }
}
```

*top level…*

```
  function getCustomerData() {return customerData;}
  function getRawDataOfCustomers()    {return customerData._data;}
  function setRawDataOfCustomers(arg) {customerData = new CustomerData(arg);}
```

The most important area to deal with is the updates. So, while I look at all the callers of `getRawDataOfCustomers`, I'm focused on those where the data is changed. To remind you, here's the update again:

*sample update…*

```
getRawDataOfCustomers()[customerID].usages[year][month] = amount;
```

The general mechanics now say to return the full customer and use an accessor, creating one if needed. I don't have a setter on the customer for this update, and this one digs into the structure. So, to make one, I begin by using *Extract Function (106)* on the code that digs into the data structure.

*sample update…*

```
  setUsage(customerID, year, month, amount);
```

*top level…*

```
function setUsage(customerID, year, month, amount) {
  getRawDataOfCustomers()[customerID].usages[year][month] = amount;
}
```

I then use *Move Function (198)* to move it into the new customer data class.

*sample update…*

```
getCustomerData().setUsage(customerID, year, month, amount);
```

*class CustomerData…*

```
setUsage(customerID, year, month, amount) {
  this._data[customerID].usages[year][month] = amount;
}
```

When working with a big data structure, I like to concentrate on the updates. Getting them visible and gathered in a single place is the most important part of the encapsulation.

At some point, I will think I've got them all—but how can I be sure? There's a couple of ways to check. One is to modify `getRawDataOfCustomers` to return a deep copy of the data; if my test coverage is good, one of the tests should break if I missed a modification.

*top level…*

```
function getCustomerData() {return customerData;}
function getRawDataOfCustomers()    {return customerData.rawData;}
function setRawDataOfCustomers(arg) {customerData = new CustomerData(arg);}
```

*class CustomerData…*

```
  get rawData() {
    return _.cloneDeep(this._data);
  }
```

I'm using the lodash library to make a deep copy.

Another approach is to return a read-only proxy for the data structure. Such a proxy could raise an exception if the client code tries to modify the underlying object. Some languages make this easy, but it's a pain in JavaScript, so I'll leave it as an exercise for the reader. I could also take a copy and recursively freeze it to detect any modifications.

Dealing with the updates is valuable, but what about the readers? Here there are a few options.

The first option is to do the same thing as I did for the setters. Extract all the reads into their own functions and move them into the customer data class.

*class CustomerData…*

```
  usage(customerID, year, month) {
    return this._data[customerID].usages[year][month];
  }
```

*top level…*

```
function compareUsage (customerID, laterYear, month) {
  const later   = getCustomerData().usage(customerID, laterYear, month);
  const earlier = getCustomerData().usage(customerID, laterYear - 1, month);
  return {laterAmount: later, change: later - earlier};
}
```

The great thing about this approach is that it gives `customerData` an explicit API that captures all the uses made of it. I can look at the class and see all their uses of the data. But this can be a lot of code for lots of special cases. Modern languages provide good affordances for digging into a list-and-hash [mf-lh] data structure, so it's useful to give clients just such a data structure to work with.

If the client wants a data structure, I can just hand out the actual data. But the problem with this is that there's no way to prevent clients from modifying the data directly, which breaks the whole point of encapsulating all the updates inside functions. Consequently, the simplest thing to do is to provide a copy of the underlying data, using the `rawData` method I wrote earlier.

*class CustomerData…*

```
  get rawData() {
    return _.cloneDeep(this._data);
  }
```

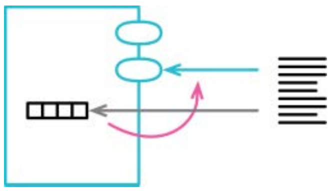
*top level…*

```
function compareUsage (customerID, laterYear, month) {
  const later   = getCustomerData().rawData[customerID].usages[laterYear][month];
  const earlier = getCustomerData().rawData[customerID].usages[laterYear - 1][month];
  return {laterAmount: later, change: later - earlier};
}
```

But although it's simple, there are downsides. The most obvious problem is the cost of copying a large data structure, which may turn out to be a performance problem. As with anything like this, however, the performance cost might be acceptable—I would want to measure its impact before I start to worry about it. There may also be confusion if clients expect modifying the copied data to modify the original. In those cases, a read-only proxy or freezing the copied data might provide a helpful error should they do this.

Another option is more work, but offers the most control: Apply Encapsulate Record recursively. With this, I turn the customer record into its own class, apply *Encapsulate Collection (170)* to the usages, and create a usage class. I can then enforce control of updates by using accessors, perhaps applying *Change Reference to Value (252)* on the usage objects. But this can be a lot of effort for a large data structure—and not really needed if I don't access that much of the data structure. Sometimes, a judicious mix of getters and new classes may work, using a getter to dig deep into the structure but returning an object that wraps the structure rather than the unencapsulated data. I wrote about this kind of thing in an article "Refactoring Code to Load a Document" [mf-ref-doc].

## Encapsulate Collection



```
class Person {
  get courses() {return this._courses;}
  set courses(aList) {this._courses = aList;}
```

⇓

```
class Person {
  get courses() {return this._courses.slice();}
  addCourse(aCourse)    { ... }
  removeCourse(aCourse) { ... }
```

### Motivation

I like encapsulating any mutable data in my programs. This makes it easier to see when and how data structures are modified, which then makes it easier to change those data structures when I need to. Encapsulation is often encouraged, particularly by object-oriented developers, but a common mistake occurs when working with collections. Access to a collection variable may be encapsulated, but if the getter returns the collection itself, then that collection's membership can be altered without the enclosing class being able to intervene.

To avoid this, I provide collection modifier methods—usually `add` and `remove`—on the class itself. This way, changes to the collection go through the owning class, giving me the opportunity to modify such changes as the program evolves.

Iff the team has the habit to not to modify collections outside the original module, just providing these methods may be enough. However, it's usually unwise to rely on such habits; a mistake here can lead to bugs that are difficult to track down later. A better approach is to ensure that the getter for the collection does not return the raw collection, so that clients cannot accidentally change it.

One way to prevent modification of the underlying collection is by never returning a collection value. In this approach, any use of a collection field is done with specific methods on the owning class, replacing `aCustomer.orders.size` with `aCustomer.numberOfOrders`. I don't agree with this approach. Modern languages have rich collection classes with standardized interfaces, which can be combined in useful ways such as Collection Pipelines [mf-cp]. Putting in special methods to handle this kind of functionality adds a lot of extra code and cripples the easy composability of collection operations.

Another way is to allow some form of read-only access to a collection. Java, for example, makes it easy to return a read-only proxy to the collection. Such a proxy forwards all reads to the underlying collection, but blocks all writes—in Java's case, throwing an exception. A similar route is used by libraries that base their collection composition on some kind of iterator or enumerable object—providing that iterator cannot modify the underlying collection.

Probably the most common approach is to provide a getting method for the collection, but make it return a copy of the underlying collection. That way, any modifications to the copy don't affect the encapsulated collection. This might cause some confusion if programmers expect the returned collection to modify the source field—but in many code bases, programmers are used to collection getters providing copies. If the collection is huge, this may be a performance issue—but most lists aren't all that big, so the general rules for performance should apply (*Refactoring and Performance (64)*).

Another difference between using a proxy and a copy is that a modification of the source data will be visible in the proxy but not in a copy. This isn't an issue most of the time, because lists accessed in this way are usually only held for a short time.

What's important here is consistency within a code base. Use only one mechanism so everyone can get used to how it behaves and expect it when calling any collection accessor function.

## Mechanics

- Apply *Encapsulate Variable (132)* if the reference to the collection isn't already encapsulated.
- Add functions to add and remove elements from the collection.

  If there is a setter for the collection, use *Remove Setting Method (331)* if possible. If not, make it take a copy of the provided collection.
- Run static checks.
- Find all references to the collection. If anyone calls modifiers on the collection, change them to use the new add/remove functions. Test after each change.
- Modify the getter for the collection to return a protected view on it, using a read-only proxy or a copy.
- Test.

## Example

I start with a person class that has a field for a list of courses.

*class Person…*

```
  constructor (name) {
    this._name = name;
    this._courses = [];
  }
  get name() {return this._name;}
  get courses() {return this._courses;}
  set courses(aList) {this._courses = aList;}
```

*class Course…*

```
  constructor(name, isAdvanced) {
    this._name = name;
    this._isAdvanced = isAdvanced;
  }
  get name()       {return this._name;}
  get isAdvanced() {return this._isAdvanced;}
```

Clients use the course collection to gather information on courses.

```
numAdvancedCourses = aPerson.courses
  .filter(c => c.isAdvanced)
  .length
;
```

A naive developer would say this class has proper data encapsulation: After all, each field is protected by accessor methods. But I would argue that the list of courses isn't properly encapsulated. Certainly, anyone updating the courses as a single value has proper control through the setter:

*client code…*

```
  const basicCourseNames = readBasicCourseNames(filename);
  aPerson.courses = basicCourseNames.map(name => new Course(name, false));
```

But clients might find it easier to update the course list directly.

*client code…*

```
  for(const name of readBasicCourseNames(filename)) {
    aPerson.courses.push(new Course(name, false));
  }
```

This violates encapsulating because the person class has no ability to take control when the list is updated in this way. While the reference to the field is encapsulated, the content of the field is not.

I'll begin creating proper encapsulation by adding methods to the person class that allow a client to add and remove individual courses.

*class Person...*

```
  addCourse(aCourse) {
    this._courses.push(aCourse);
  }
  removeCourse(aCourse, fnIfAbsent = () => {throw new RangeError();}) {
    const index = this._courses.indexOf(aCourse);
    if (index === -1) fnIfAbsent();
    else this._courses.splice(index, 1);
  }
```

With a removal, I have to decide what to do if a client asks to remove an element that isn't in the collection. I can either shrug, or raise an error. With this code, I default to raising an error, but give the callers an opportunity to do something else if they wish.

I then change any code that calls modifiers directly on the collection to use new methods.

*client code...*

```
for(const name of readBasicCourseNames(filename)) {
  aPerson.addCourse(new Course(name, false));
}
```

With individual add and remove methods, there is usually no need for `setCourses`, in which case I'll use *Remove Setting Method (331)* on it. Should the API need a setting method for some reason, I ensure it puts a copy of the collection in the field.

*class Person...*

```
set courses(aList) {this._courses = aList.slice();}
```

All this enables the clients to use the right kind of modifier methods, but I prefer to ensure nobody modifies the list without using them. I can do this by providing a copy.

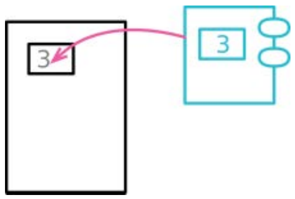
*class Person...*

```
get courses() {return this._courses.slice();}
```

In general, I find it wise to be moderately paranoid about collections and I'd rather copy them unnecessarily than debug errors due to unexpected modifications. Modifications aren't always obvious; for example, sorting an array in JavaScript modifies the original, while many languages default to making a copy for an operation that changes a collection. Any class that's responsible for managing a collection should always give out copies—but I also get into the habit of making a copy if I do something that's liable to change a collection.

## Replace Primitive with Object

formerly: *Replace Data Value with Object*
formerly: *Replace Type Code with Class*



```
orders.filter(o => "high" === o.priority
                || "rush" === o.priority);
```

⇓

```
orders.filter(o => o.priority.higherThan(new Priority("normal")))
```

### Motivation

Often, in early stages of development you make decisions about representing simple facts as simple data items, such as numbers or strings. As development proceeds, those simple items aren't so simple anymore. A telephone number may be represented as a string for a while, but later it will need special behavior for formatting, extracting the area code, and the like. This kind of logic can quickly end up being duplicated around the code base, increasing the effort whenever it needs to be used.

As soon as I realize I want to do something other than simple printing, I like to create a new class for that bit of data. At first, such a class does little more than wrap the primitive—but once I have that class, I have a place to put behavior specific to its needs. These little values start very humble, but once nurtured they can grow into useful tools. They may not look like much, but I find their effects on a code base can be surprisingly large. Indeed many experienced developers consider this to be one of the most valuable refactorings in the toolkit—even though it often seems counterintuitive to a new programmer.

## Mechanics

- Apply *Encapsulate Variable (132)* if it isn't already.
- Create a simple value class for the data value. It should take the existing value in its constructor and provide a getter for that value.
- Run static checks.
- Change the setter to create a new instance of the value class and store that in the field, changing the type of the field if present.
- Change the getter to return the result of invoking the getter of the new class.
- Test.
- Consider using *Rename Function (124)* on the original accessors to better reflect what they do.
- Consider clarifying the role of the new object as a value or reference object by applying *Change Reference to Value (252)* or *Change Value to Reference (256)*.

## Example

I begin with a simple order class that reads its data from a simple record structure. One of its properties is a priority, which it reads as a simple string.

*class Order...*

```
constructor(data) {
  this.priority = data.priority;
  // more initialization
```

Some client codes uses it like this:

*client...*

```
highPriorityCount = orders.filter(o => "high" === o.priority
                                    || "rush" === o.priority)
                          .length;
```

Whenever I'm fiddling with a data value, the first thing I do is use *Encapsulate Variable (132)* on it.

*class Order...*

```
  get priority()        {return this._priority;}
  set priority(aString) {this._priority = aString;}
```

The constructor line that initializes the priority will now use the setter I define here.

This self-encapsulates the field so I can preserve its current use while I manipulate the data itself.

I create a simple value class for the priority. It has a constructor for the value and a conversion function to return a string.

```
class Priority {
  constructor(value) {this._value = value;}
  toString() {return this._value;}
}
```

I prefer using a conversion function (toString) rather than a getter (value) here. For clients of the class, asking for the string representation should feel more like a conversion than getting a property.

I then modify the accessors to use this new class.

*class Order...*

```
get priority()        {return this._priority.toString();}
set priority(aString) {this._priority = new Priority(aString);}
```

Now that I have a priority class, I find the current getter on the order to be misleading. It doesn't return the priority—but a string that describes the priority. My immediate move is to use *Rename Function (124)*.

*class Order...*

```
get priorityString()  {return this._priority.toString();}
set priority(aString) {this._priority = new Priority(aString);}
```

*client...*

```
highPriorityCount = orders.filter(o => "high" === o.priorityString
                                    || "rush" === o.priorityString)
                          .length;
```

In this case, I'm happy to retain the name of the setter. The name of the argument communicates what it expects.

Now I'm done with the formal refactoring. But as I look at who uses the priority, I consider whether they should use the priority class themselves. As a result, I provide a getter on order that provides the new priority object directly.

*class Order...*

```
  get priority()        {return this._priority;}
  get priorityString()  {return this._priority.toString();}
  set priority(aString) {this._priority = new Priority(aString);}
```

*client...*

```
highPriorityCount = orders.filter(o => "high" === o.priority.toString()
                                    || "rush" === o.priority.toString())
                          .length;
```

As the priority class becomes useful elsewhere, I would allow clients of the order to use the setter with a priority instance, which I do by adjusting the priority constructor.

*class Priority...*

```
constructor(value) {
  if (value instanceof Priority) return value;
  this._value = value;
}
```

The point of all this is that now, my new priority class can be useful as a place for new behavior—either new to the code or moved from elsewhere. Here's some simple code to add validation of priority values and comparison logic:

*class Priority...*

```
constructor(value) {
  if (value instanceof Priority) return value;
  if (Priority.legalValues().includes(value))
    this._value = value;
  else
    throw new Error(`<${value}> is invalid for Priority`);
}
toString() {return this._value;}
get _index() {return Priority.legalValues().findIndex(s => s === this._value);}
static legalValues() {return ['low', 'normal', 'high', 'rush'];}

equals(other) {return this._index === other._index;}
higherThan(other) {return this._index > other._index;}
lowerThan(other) {return this._index < other._index;}
```

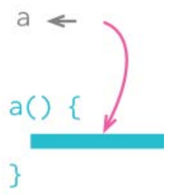
As I do this, I decide that a priority should be a value object, so I provide an equals method and ensure that it is immutable.

Now I've added that behavior, I can make the client code more meaningful:

*client...*

```
highPriorityCount = orders.filter(o => o.priority.higherThan(new Priority("normal")))
                          .length;
```

## Replace Temp with Query



```
const basePrice = this._quantity * this._itemPrice;
if (basePrice > 1000)
  return basePrice * 0.95;
else
  return basePrice * 0.98;
```

⇓

```
get basePrice() {this._quantity * this._itemPrice;}

...

if (this.basePrice > 1000)
  return this.basePrice * 0.95;
else
  return this.basePrice * 0.98;
```

### Motivation

One use of temporary variables is to capture the value of some code in order to refer to it later in a function. Using a temp allows me to refer to the value while explaining its meaning and avoiding repeating the code that calculates it. But while using a variable is handy, it can often be worthwhile to go a step further and use a function instead.

If I'm working on breaking up a large function, turning variables into their own functions makes it easier to extract parts of the function, since I no longer need to pass in variables into the extracted functions. Putting this logic into functions often also sets up a stronger boundary between the extracted logic and the original function, which helps me spot and avoid awkward dependencies and side effects.

Using functions instead of variables also allows me to avoid duplicating the calculation logic in similar functions. Whenever I see variables calculated in the same way in different places, I look to turn them into a single function.

This refactoring works best if I'm inside a class, since the class provides a shared context for the methods I'm extracting. Outside of a class, I'm liable to have too many parameters in a top-level function which negates much of the benefit of using a function. Nested functions can avoid this, but they limit my ability to share the logic between related functions.

Only some temporary variables are suitable for Replace Temp with Query. The variable needs to be calculated once and then only be read afterwards. In the simplest case, this means the variable is assigned to once, but it's also possible to have several assignments in a more complicated lump of code—all of which has to be extracted into the query. Furthermore, the logic used to calculate the variable must yield the same result when the variable is used later—which rules out variables used as snapshots with names like `oldAddress`.

## Mechanics

- Check that the variable is determined entirely before it's used, and the code that calculates it does not yield a different value whenever it is used.
- If the variable isn't read-only, and can be made read-only, do so.
- Test.
- Extract the assignment of the variable into a function.

  If the variable and the function cannot share a name, use a temporary name for the function.

  Ensure the extracted function is free of side effects. If not, use *Separate Query from Modifier (306)*.
- Test.
- Use *Inline Variable (123)* to remove the temp.

## Example

Here is a simple class:

*class Order…*

```
  constructor(quantity, item) {
    this._quantity = quantity;
    this._item = item;
  }

  get price() {
    var basePrice = this._quantity * this._item.price;
    var discountFactor = 0.98;
    if (basePrice > 1000) discountFactor -= 0.03;
    return basePrice * discountFactor;
  }
}
```

I want to replace the temps `basePrice` and `discountFactor` with methods.

Starting with `basePrice`, I make it `const` and run tests. This is a good way of checking that I haven't missed a reassignment—unlikely in such a short function but common when I'm dealing with something larger.

*class Order…*

```
  constructor(quantity, item) {
    this._quantity = quantity;
    this._item = item;
  }

  get price() {
    const basePrice = this._quantity * this._item.price;
    var discountFactor = 0.98;
    if (basePrice > 1000) discountFactor -= 0.03;
    return basePrice * discountFactor;
  }
}
```

I then extract the right-hand side of the assignment to a getting method.

*class Order…*

```
  get price() {
    const basePrice = this.basePrice;
    var discountFactor = 0.98;
    if (basePrice > 1000) discountFactor -= 0.03;
    return basePrice * discountFactor;
  }

   get basePrice() {
     return this._quantity * this._item.price;
   }
```

I test, and apply *Inline Variable (123)*.

*class Order…*

```
get price() {
  ~~const basePrice = this.basePrice;~~
  var discountFactor = 0.98;
  if (this.basePrice > 1000) discountFactor -= 0.03;
  return this.basePrice * discountFactor;
}
```

I then repeat the steps with `discountFactor`, first using *Extract Function (106)*.

*class Order…*

```
get price() {
  const discountFactor = this.discountFactor;
  return this.basePrice * discountFactor;
}

 get discountFactor() {
   var discountFactor = 0.98;
   if (this.basePrice > 1000) discountFactor -= 0.03;
   return discountFactor;
 }
```

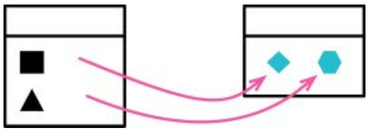
In this case I need my extracted function to contain both assignments to `discountFactor`. I can also set the original variable to be `const`.

Then, I inline:

```
get price() {
  return this.basePrice * this.discountFactor;
}
```

## Extract Class

inverse of: *Inline Class (186)*



```
class Person {
  get officeAreaCode() {return this._officeAreaCode;}
  get officeNumber()   {return this._officeNumber;}
```

⇓

```
class Person {
  get officeAreaCode() {return this._telephoneNumber.areaCode;}
  get officeNumber()   {return this._telephoneNumber.number;}
}
class TelephoneNumber {
  get areaCode() {return this._areaCode;}
  get number()   {return this._number;}
}
```

### Motivation

You've probably read guidelines that a class should be a crisp abstraction, only handle a few clear responsibilities, and so on. In practice, classes grow. You add some operations here, a bit of data there. You add a responsibility to a class feeling that it's not worth a separate class—but as that responsibility grows and breeds, the class becomes too complicated. Soon, your class is as crisp as a microwaved duck.

Imagine a class with many methods and quite a lot of data. A class that is too big to understand easily. You need to consider where it can be split—and split it. A good sign is when a subset of the data and a subset of the methods seem to go together. Other good signs are subsets of data that usually change together or are particularly dependent on each other. A useful test is to ask yourself what would happen if you remove a piece of data or a method. What other fields and methods would become nonsense?

One sign that often crops up later in development is the way the class is subtyped. You may find that subtyping affects only a few features or that some features need to be subtyped one way and other features a different way.

## Mechanics

- Decide how to split the responsibilities of the class.
- Create a new child class to express the split-off responsibilities.

  If the responsibilities of the original parent class no longer match its name, rename the parent.
- Create an instance of the child class when constructing the parent and add a link from parent to child.
- Use *Move Field (207)* on each field you wish to move. Test after each move.
- Use *Move Function (198)* to move methods to the new child. Start with lower-level methods (those being called rather than calling). Test after each move.
- Review the interfaces of both classes, remove unneeded methods, change names to better fit the new circumstances.
- Decide whether to expose the new child. If so, consider applying *Change Reference to Value (252)* to the child class.

## Example

I start with a simple person class:

*class Person...*

```
get name()    {return this._name;}
set name(arg) {this._name = arg;}
get telephoneNumber() {return `(${this.officeAreaCode}) ${this.officeNumber}`;}
get officeAreaCode()    {return this._officeAreaCode;}
set officeAreaCode(arg) {this._officeAreaCode = arg;}
get officeNumber() {return this._officeNumber;}
set officeNumber(arg) {this._officeNumber = arg;}
```

Here. I can separate the telephone number behavior into its own class. I start by defining an empty telephone number class:

```
class TelephoneNumber {
}
```

That was easy! Next, I create an instance of telephone number when constructing the person:

*class Person...*

```
constructor() {
  this._telephoneNumber = new TelephoneNumber();
}
```

*class TelephoneNumber…*

```
  get officeAreaCode()    {return this._officeAreaCode;}
  set officeAreaCode(arg) {this._officeAreaCode = arg;}
```

I then use *Move Field (207)* on one of the fields.

*class Person…*

```
get officeAreaCode()    {return this._telephoneNumber.officeAreaCode;}
set officeAreaCode(arg) {this._telephoneNumber.officeAreaCode = arg;}
```

I test, then move the next field.

*class TelephoneNumber…*

```
  get officeNumber() {return this._officeNumber;}
  set officeNumber(arg) {this._officeNumber = arg;}
```

*class Person…*

```
get officeNumber() {return this._telephoneNumber.officeNumber;}
set officeNumber(arg) {this._telephoneNumber.officeNumber = arg;}
```

Test again, then move the telephone number method.

*class TelephoneNumber…*

```
  get telephoneNumber() {return `(${this.officeAreaCode}) ${this.officeNumber}`;}
```

*class Person…*

```
get telephoneNumber() {return this._telephoneNumber.telephoneNumber;}
```

Now I should tidy things up. Having "office" as part of the telephone number code makes no sense, so I rename them.

*class TelephoneNumber…*

```
get areaCode()    {return this._areaCode;}
set areaCode(arg) {this._areaCode = arg;}

get number()    {return this._number;}
set number(arg) {this._number = arg;}
```

*class Person…*

```
get officeAreaCode()    {return this._telephoneNumber.areaCode;}
set officeAreaCode(arg) {this._telephoneNumber.areaCode = arg;}
get officeNumber()    {return this._telephoneNumber.number;}
set officeNumber(arg) {this._telephoneNumber.number = arg;}
```

The telephone number method on the telephone number class also doesn't make much sense, so I apply *Rename Function (124)*.

*class TelephoneNumber…*

```
toString() {return `(${this.areaCode}) ${this.number}`;}
```

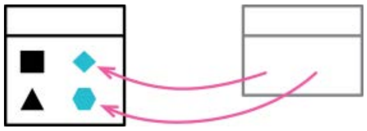
*class Person…*

```
get telephoneNumber() {return this._telephoneNumber.toString();}
```

Telephone numbers are generally useful, so I think I'll expose the new object to clients. I can replace those "office" methods with accessors for the telephone number. But this way, the telephone number will work better as a Value Object [mf-vo], so I would apply *Change Reference to Value (252)* first (that refactoring's example shows how I'd do that for the telephone number).

## Inline Class

inverse of: *Extract Class (182)*



```
class Person {
  get officeAreaCode() {return this._telephoneNumber.areaCode;}
  get officeNumber()   {return this._telephoneNumber.number;}
}
class TelephoneNumber {
  get areaCode() {return this._areaCode;}
  get number()   {return this._number;}
}
```

⇓

```
class Person {
  get officeAreaCode() {return this._officeAreaCode;}
  get officeNumber()   {return this._officeNumber;}
```

### Motivation

Inline Class is the inverse of *Extract Class (182)*. I use Inline Class if a class is no longer pulling its weight and shouldn't be around any more. Often, this is the result of refactoring that moves other responsibilities out of the class so there is little left. At that point, I fold the class into another—one that makes most use of the runt class.

Another reason to use Inline Class is if I have two classes that I want to refactor into a pair of classes with a different allocation of features. I may find it easier to first use Inline Class to combine them into a single class, then *Extract Class (182)* to make the new separation. This is a general approach when reorganizing things: Sometimes, it's easier to move elements one at a time from one context to another, but sometimes it's better to use an inline refactoring to collapse the contexts together, then use an extract refactoring to separate them into different elements.

## Mechanics

- In the target class, create functions for all the public functions of the source class. These functions should just delegate to the source class.
- Change all references to source class methods so they use the target class's delegators instead. Test after each change.
- Move all the functions and data from the source class into the target, testing after each move, until the source class is empty.
- Delete the source class and hold a short, simple funeral service.

## Example

Here's a class that holds a couple of pieces of tracking information for a shipment.

```
class TrackingInformation {
  get shippingCompany()    {return this._shippingCompany;}
  set shippingCompany(arg) {this._shippingCompany = arg;}
  get trackingNumber()    {return this._trackingNumber;}
  set trackingNumber(arg) {this._trackingNumber = arg;}
  get display()            {
    return `${this.shippingCompany}: ${this.trackingNumber}`;
  }
}
```

It's used as part of a shipment class.

*class Shipment…*

```
get trackingInfo() {
  return this._trackingInformation.display;
}
get trackingInformation()    {return this._trackingInformation;}
set trackingInformation(aTrackingInformation) {
  this._trackingInformation = aTrackingInformation;
}
```

While this class may have been worthwhile in the past, I no longer feel it's pulling its weight, so I want to inline it into `Shipment`.

I start by looking at places that are invoking the methods of `TrackingInformation`.

*caller…*

```
aShipment.trackingInformation.shippingCompany = request.vendor;
```

I'm going to move all such functions to `Shipment`, but I do it slightly differently to how I usually do *Move Function (198)*. In this case, I start by putting a delegating method into the shipment, and adjusting the client to call that.

*class Shipment…*

```
  set shippingCompany(arg) {this._trackingInformation.shippingCompany = arg;}
```

*caller…*

```
aShipment.~~trackingInformation.~~shippingCompany = request.vendor;
```

I do this for all the elements of tracking information that are used by clients. Once I've done that, I can move all the elements of the tracking information over into the shipment class.

I start by applying *Inline Function (115)* to the `display` method.

*class Shipment…*

```
get trackingInfo() {
  return `${this.shippingCompany}: ${this.trackingNumber}`;
}
```

I move the shipping company field.

```
get shippingCompany()    {return this.~~_trackingInformation.~~_shippingCompany;}
set shippingCompany(arg) {this.~~_trackingInformation.~~_shippingCompany = arg;}
```

I don't use the full mechanics for *Move Field (207)* since in this case I only reference `shippingCompany` from `Shipment` which is the target of the move. I thus don't need the steps that put a reference from the source to the target.

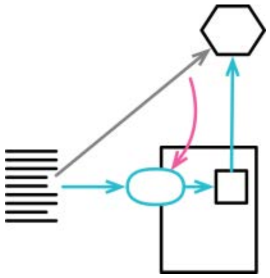
I continue until everything is moved over. Once I've done that, I can delete the tracking information class.

*class Shipment…*

```
get trackingInfo() {
  return `${this.shippingCompany}: ${this.trackingNumber}`;
}
get shippingCompany()    {return this._shippingCompany;}
set shippingCompany(arg) {this._shippingCompany = arg;}
get trackingNumber()    {return this._trackingNumber;}
set trackingNumber(arg) {this._trackingNumber = arg;}
```

## Hide Delegate

inverse of: *Remove Middle Man (192)*



```
manager = aPerson.department.manager;
```

⇓

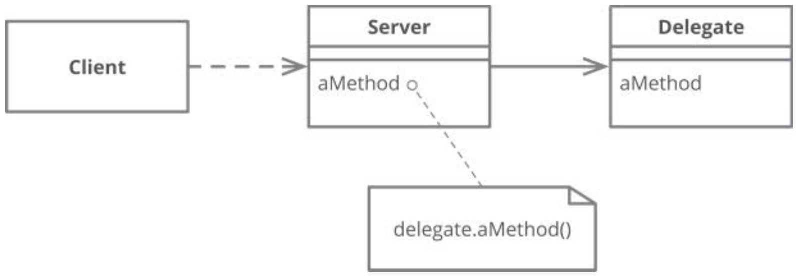
```
manager = aPerson.manager;

class Person {
  get manager() {return this.department.manager;}
```

### Motivation

One of the keys—if not *the* key—to good modular design is encapsulation. Encapsulation means that modules need to know less about other parts of the system. Then, when things change, fewer modules need to be told about the change—which makes the change easier to make.

When we are first taught about object orientation, we are told that encapsulation means hiding our fields. As we become more sophisticated, we realize there is more that we can encapsulate.

If I have some client code that calls a method defined on an object in a field of a server object, the client needs to know about this delegate object. If the delegate changes its interface, changes propagate to all the clients of the server that use the delegate. I can remove this dependency by placing a simple delegating method on the server that hides the delegate. Then any changes I make to the delegate propagate only to the server and not to the clients.



## Mechanics

- For each method on the delegate, create a simple delegating method on the server.
- Adjust the client to call the server. Test after each change.
- If no client needs to access the delegate anymore, remove the server's accessor for the delegate.
- Test.

## Example

I start with a person and a department.

*class Person…*

```
constructor(name) {
  this._name = name;
}
get name() {return this._name;}
get department()    {return this._department;}
set department(arg) {this._department = arg;}
```

*class Department…*

```
get chargeCode()    {return this._chargeCode;}
set chargeCode(arg) {this._chargeCode = arg;}
get manager()    {return this._manager;}
set manager(arg) {this._manager = arg;}
```

Some client code wants to know the manager of a person. To do this, it needs to get the department first.

*client code…*

```
manager = aPerson.department.manager;
```

This reveals to the client how the department class works and that the department is responsible for tracking the manager. I can reduce this coupling by hiding the department class from the client. I do this by creating a simple delegating method on person:

*class Person…*

```
  get manager() {return this._department.manager;}
```

I now need to change all clients of person to use this new method:

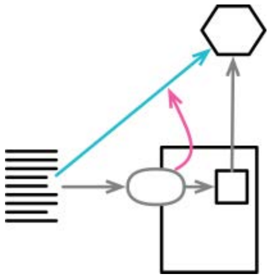
*client code…*

```
manager = aPerson~~.department~~.manager;
```

Once I've made the change for all methods of department and for all the clients of person, I can remove the `department` accessor on person.

## Remove Middle Man

inverse of: *Hide Delegate (189)*



```
manager = aPerson.manager;

class Person {
  get manager() {return this.department.manager;}
```

⇓

```
manager = aPerson.department.manager;
```

### Motivation

In the motivation for *Hide Delegate (189)*, I talked about the advantages of encapsulating the use of a delegated object. There is a price for this. Every time the client wants to use a new feature of the delegate, I have to add a simple delegating method to the server. After adding features for a while, I get irritated with all this forwarding. The server class is just a middle man (*Middle Man (81)*), and perhaps it's time for the client to call the delegate directly. (This smell often pops up when people get overenthusiastic about following the Law of Demeter, which I'd like a lot more if it were called the Occasionally Useful Suggestion of Demeter.)

It's hard to figure out what the right amount of hiding is. Fortunately, with *Hide Delegate (189)* and Remove Middle Man, it doesn't matter so much. I can adjust my code as time goes on. As the system changes, the basis for how much I hide also changes. A good encapsulation six months ago may be awkward now. Refactoring means I never have to say I'm sorry—I just fix it.

## Mechanics

- Create a getter for the delegate.
- For each client use of a delegating method, replace the call to the delegating method by chaining through the accessor. Test after each replacement.

  If all calls to a delegating method are replaced, you can delete the delegating method.

  With automated refactorings, you can use *Encapsulate Variable (132)* on the delegate field and then *Inline Function (115)* on all the methods that use it.

## Example

I begin with a person class that uses a linked department object to determine a manager. (If you're reading this book sequentially, this example may look eerily familiar.)

*client code...*

```
manager = aPerson.manager;
```

*class Person...*

```
get manager() {return this._department.manager;}
```

*class Department...*

```
get manager()    {return this._manager;}
```

This is simple to use and encapsulates the department. However, if lots of methods are doing this, I end up with too many of these simple delegations on the person. That's when it is good to remove the middle man. First, I make an accessor for the delegate:

*class Person...*

```
  get department()    {return this._department;}
```

Now I go to each client at a time and modify them to use the department directly.

*client code...*

```
manager = aPerson.department.manager;
```

Once I've done this with all the clients, I can remove the manager method from `Person`. I can repeat this process for any other simple delegations on `Person`.

I can do a mixture here. Some delegations may be so common that I'd like to keep them to make client code easier to work with. There is no absolute reason why I should either hide a delegate or remove a middle man—particular circumstances suggest which approach to take, and reasonable people can differ on what works best.

If I have automated refactorings, then there's a useful variation on these steps. First, I use *Encapsulate Variable (132)* on `department`. This changes the manager getter to use the public department getter:

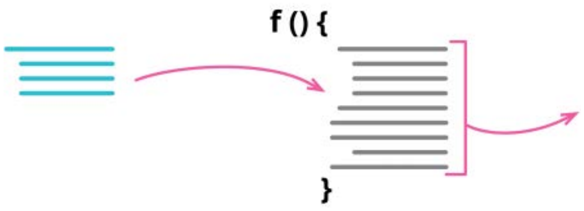
*class Person…*

```
get manager() {return this.department.manager;}
```

*The change is rather too subtle in JavaScript, but by removing the underscore from* `department` *I'm using the new getter rather than accessing the field directly.*

Then I apply *Inline Function (115)* on the manager method to replace all the callers at once.

## Substitute Algorithm



```
function foundPerson(people) {
  for(let i = 0; i < people.length; i++) {
    if (people[i] === "Don") {
      return "Don";
    }
    if (people[i] === "John") {
      return "John";
    }
    if (people[i] === "Kent") {
      return "Kent";
    }
  }
  return "";
}
```

⇓

```
function foundPerson(people) {
  const candidates = ["Don", "John", "Kent"];
  return people.find(p => candidates.includes(p)) || '';
}
```

### Motivation

I've never tried to skin a cat. I'm told there are several ways to do it. I'm sure some are easier than others. So it is with algorithms. If I find a clearer way to do something, I replace the complicated way with the clearer way. Refactoring can break down something complex into simpler pieces, but sometimes I just reach the point at which I have to remove the whole algorithm and replace it with something simpler. This occurs as I learn more about the problem and realize that there's an easier way to do it. It also happens if I start using a library that supplies features that duplicate my code.

Sometimes, when I want to change the algorithm to work slightly differently, it's easier to start by replacing it with something that would make my change more straightforward to make.

When I have to take this step, I have to be sure I've decomposed the method as much as I can. Replacing a large, complex algorithm is very difficult; only by making it simple can I make the substitution tractable.

## Mechanics

- Arrange the code to be replaced so that it fills a complete function.
- Prepare tests using this function only, to capture its behavior.
- Prepare your alternative algorithm.
- Run static checks.
- Run tests to compare the output of the old algorithm to the new one. If they are the same, you're done. Otherwise, use the old algorithm for comparison in testing and debugging.

# Chapter 8

# Moving Features

So far, the refactorings have been about creating, removing, and renaming program elements. Another important part of refactoring is moving elements between contexts. I use *Move Function (198)* to move functions between classes and other modules. Fields can move too, with *Move Field (207)*.

I also move individual statements around. I use *Move Statements into Function (213)* and *Move Statements to Callers (217)* to move them in or out of functions, as well as *Slide Statements (223)* to move them within a function. Sometimes, I can take some statements that match an existing function and use *Replace Inline Code with Function Call (222)* to remove the duplication.

Two refactorings I often do with loops are *Split Loop (227)*, to ensure a loop does only one thing, and *Replace Loop with Pipeline (231)* to get rid of a loop entirely.

And then there's the favorite refactoring of many a fine programmer: *Remove Dead Code (237)*. Nothing is as satisfying as applying the digital flamethrower to superfluous statements.

## Move Function

formerly: *Move Method*

```
class Account {
  get overdraftCharge() {...}
```

⇓

```
class AccountType {
    get overdraftCharge() {...}
```

### Motivation

The heart of a good software design is its modularity—which is my ability to make most modifications to a program while only having to understand a small part of it. To get this modularity, I need to ensure that related software elements are grouped together and the links between them are easy to find and understand. But my understanding of how to do this isn't static—as I better understand what I'm doing, I learn how to best group together software elements. To reflect that growing understanding, I need to move elements around.

All functions live in some context; it may be global, but usually it's some form of a module. In an object-oriented program, the core modular context is a class. Nesting a function within another creates another common context. Different languages provide varied forms of modularity, each creating a context for a function to live in.

One of the most straightforward reasons to move a function is when it references elements in other contexts more than the one it currently resides in. Moving it together with those elements often improves encapsulation, allowing other parts of the software to be less dependent on the details of this module.

Similarly, I may move a function because of where its callers live, or where I need to call it from in my next enhancement. A function defined as a helper inside another function may have value on its own, so it's worth moving it to somewhere more accessible. A method on a class may be easier for me to use if shifted to another.

Deciding to move a function is rarely an easy decision. To help me decide, I examine the current and candidate contexts for that function. I need to look at what functions call this one, what functions are called by the moving function, and what data that function uses. Often, I see that I need a new context for a group of functions and create one with *Combine Functions into Class (144)* or *Extract Class (182)*. Although it can be difficult to decide where the best place for a function is, the more difficult this choice, often the less it matters. I find it valuable to try working with functions in one context, knowing I'll learn how well they fit, and if they don't fit I can always move them later.

## Mechanics

- Examine all the program elements used by the chosen function in its current context. Consider whether they should move too.

  If I find a called function that should also move, I usually move it first. That way, moving a clusters of functions begins with the one that has the least dependency on the others in the group.

  If a high-level function is the only caller of subfunctions, then you can inline those functions into the high-level method, move, and reextract at the destination.

- Check if the chosen function is a polymorphic method.

  If I'm in an object-oriented language, I have to take account of super- and subclass declarations.

- Copy the function to the target context. Adjust it to fit in its new home.

  If the body uses elements in the source context, I need to either pass those elements as parameters or pass a reference to that source context.

  Moving a function often means I need to come up with a different name that works better in the new context.

- Perform static analysis.
- Figure out how to reference the target function from the source context.
- Turn the source function into a delegating function.
- Test.
- Consider *Inline Function (115)* on the source function.

  The source function can stay indefinitely as a delegating function. But if its callers can just as easily reach the target directly, then it's better to remove the middle man.

## Example: Moving a Nested Function to Top Level

I'll begin with a function that calculates the total distance for a GPS track record.

```
function trackSummary(points) {
  const totalTime = calculateTime();
  const totalDistance = calculateDistance();
  const pace = totalTime / 60 /  totalDistance ;
  return {
    time: totalTime,
    distance: totalDistance,
    pace: pace
  };

  function calculateDistance() {
    let result = 0;
    for (let i = 1; i < points.length; i++) {
      result += distance(points[i-1], points[i]);
    }
    return result;
  }

 function distance(p1,p2) { ... }
 function radians(degrees) { ... }
 function calculateTime() { ... }

}
```

I'd like to move `calculateDistance` to the top level so I can calculate distances for tracks without all the other parts of the summary.

I begin by copying the function to the top level.

```
function trackSummary(points) {
  const totalTime = calculateTime();
  const totalDistance = calculateDistance();
  const pace = totalTime / 60 /  totalDistance ;
  return {
    time: totalTime,
    distance: totalDistance,
    pace: pace
  };

  function calculateDistance() {
    let result = 0;
    for (let i = 1; i < points.length; i++) {
      result += distance(points[i-1], points[i]);
    }
    return result;
  }
  ...
```

```
    function distance(p1,p2) { ... }
    function radians(degrees) { ... }
    function calculateTime() { ... }

  }

  function top_calculateDistance() {
    let result = 0;
    for (let i = 1; i < points.length; i++) {
      result += distance(points[i-1], points[i]);
    }
    return result;
  }
```

When I copy a function like this, I like to change the name so I can distinguish them both in the code and in my head. I don't want to think about what the right name should be right now, so I create a temporary name.

The program still works, but my static analysis is rightly rather upset. The new function has two undefined symbols: `distance` and `points`. The natural way to deal with `points` is to pass it in as a parameter.

```
function top_calculateDistance(points) {
  let result = 0;
  for (let i = 1; i < points.length; i++) {
    result += distance(points[i-1], points[i]);
  }
  return result;
}
```

I could do the same with `distance`, but perhaps it makes sense to move it together with `calculateDistance`. Here's the relevant code:

*function trackSummary…*

```
function distance(p1,p2) {
  // haversine formula see http://www.movable-type.co.uk/scripts/latlong.html
  const EARTH_RADIUS = 3959; // in miles
  const dLat = radians(p2.lat) - radians(p1.lat);
  const dLon = radians(p2.lon) - radians(p1.lon);
  const a = Math.pow(Math.sin(dLat / 2),2)
          + Math.cos(radians(p2.lat))
          * Math.cos(radians(p1.lat))
          * Math.pow(Math.sin(dLon / 2), 2);
  const c = 2 * Math.atan2(Math.sqrt(a), Math.sqrt(1-a));
  return EARTH_RADIUS * c;
}
function radians(degrees) {
  return degrees * Math.PI / 180;
}
```

I can see that `distance` only uses `radians` and `radians` doesn't use anything inside its current context. So rather than pass the functions, I might as well move them too. I can make a small step in this direction by moving them from their current context to nest them inside the nested `calculateDistance`.

```
function trackSummary(points) {
  const totalTime = calculateTime();
  const totalDistance = calculateDistance();
  const pace = totalTime / 60 /  totalDistance ;
  return {
    time: totalTime,
    distance: totalDistance,
    pace: pace
  };

  function calculateDistance() {
    let result = 0;
    for (let i = 1; i < points.length; i++) {
      result += distance(points[i-1], points[i]);
    }
    return result;

   function distance(p1,p2) { ... }
   function radians(degrees) { ... }

 }
```

By doing this, I can use both static analysis and testing to tell me if there are any complications. In this case all is well, so I can copy them over to `top_calculateDistance`.

```
function top_calculateDistance(points) {
  let result = 0;
  for (let i = 1; i < points.length; i++) {
    result += distance(points[i-1], points[i]);
  }
  return result;

 function distance(p1,p2) { ... }
 function radians(degrees) { ... }

}
```

Again, the copy doesn't change how the program runs, but does give me an opportunity for more static analysis. Had I not spotted that `distance` calls `radians`, the linter would have caught it at this step.

Now that I have prepared the table, it's time for the major change—the body of the original `calculateDistance` will now call `top_calculateDistance`:

```
function trackSummary(points) {
  const totalTime = calculateTime();
  const totalDistance = calculateDistance();
  const pace = totalTime / 60 /  totalDistance ;
  return {
    time: totalTime,
    distance: totalDistance,
    pace: pace
  };

  function calculateDistance() {
    return top_calculateDistance(points);
  }
```

This is the crucial time to run tests to fully test that the moved function has bedded down in its new home.

With that done, it's like unpacking the boxes after moving house. The first thing is to decide whether to keep the original function that's just delegating or not. In this case, there are few callers and, as usual with nested functions, they are highly localized. So I'm happy to get rid of it.

```
function trackSummary(points) {
  const totalTime = calculateTime();
  const totalDistance = top_calculateDistance(points);
  const pace = totalTime / 60 /  totalDistance ;
  return {
    time: totalTime,
    distance: totalDistance,
    pace: pace
  };
```

Now is also a good time to think about what I want the name to be. Since the top-level function has the highest visibility, I'd like it to have the best name. `totalDistance` seems like a good choice. I can't use that immediately since it will be shadowed by the variable inside `trackSummary`—but I don't see any reason to keep that anyway, so I use *Inline Variable (123)* on it.

```
function trackSummary(points) {
  const totalTime = calculateTime();
  const pace = totalTime / 60 /  totalDistance(points) ;
  return {
    time: totalTime,
    distance: totalDistance(points),
    pace: pace
  };
```

```
function totalDistance(points) {
  let result = 0;
  for (let i = 1; i < points.length; i++) {
    result += distance(points[i-1], points[i]);
  }
  return result;
```

If I'd had the need to keep the variable, I'd have renamed it to something like `totalDistanceCache` or `distance`.

Since the functions for `distance` and `radians` don't depend on anything inside `totalDistance`, I prefer to move them to top level too, putting all four functions at the top level.

```
function trackSummary(points) { ... }
function totalDistance(points) { ... }
function distance(p1,p2) { ... }
function radians(degrees) { ... }
```

Some people would prefer to keep `distance` and `radians` inside `totalDistance` in order to restrict their visibility. In some languages that may be a consideration, but with ES 2015, JavaScript has an excellent module mechanism that's the best tool for controlling function visibility. In general, I'm wary of nested functions—they too easily set up hidden data interrelationships that can get hard to follow.

## Example: Moving between Classes

To illustrate this variety of Move Function, I'll start here:

*class Account…*

```
get bankCharge() {
  let result = 4.5;
  if (this._daysOverdrawn > 0) result += this.overdraftCharge;
  return result;
}

get overdraftCharge() {
  if (this.type.isPremium) {
    const baseCharge = 10;
    if (this.daysOverdrawn <= 7)
      return baseCharge;
    else
      return baseCharge + (this.daysOverdrawn - 7) * 0.85;
  }
  else
    return this.daysOverdrawn * 1.75;
}
```

Coming up are changes that lead to different types of account having different algorithms for determining the charge. Thus it seems natural to move `overdraftCharge` to the account type class.

The first step is to look at the features that the overdraftCharge method uses and consider whether it is worth moving a batch of methods together. In this case I need the daysOverdrawn method to remain on the account class, because that will vary with individual accounts.

Next, I copy the method body over to the account type and get it to fit.

*class AccountType...*

```
overdraftCharge(daysOverdrawn) {
  if (this.isPremium) {
    const baseCharge = 10;
    if (daysOverdrawn <= 7)
      return baseCharge;
    else
      return baseCharge + (daysOverdrawn - 7) * 0.85;
  }
  else
    return daysOverdrawn * 1.75;
}
```

In order to get the method to fit in its new location, I need to deal with two call targets that change their scope. isPremium is now a simple call on this. With daysOverdrawn I have to decide—do I pass the value or do I pass the account? For the moment, I just pass the simple value but I may well change this in the future if I require more than just the days overdrawn from the account—especially if what I want from the account varies with the account type.

Next, I replace the original method body with a delegating call.

*class Account...*

```
get bankCharge() {
  let result = 4.5;
  if (this._daysOverdrawn > 0) result += this.overdraftCharge;
  return result;
}

get overdraftCharge() {
  return this.type.overdraftCharge(this.daysOverdrawn);
}
```

Then comes the decision of whether to leave the delegation in place or to inline overdraftCharge. Inlining results in:

*class Account...*

```
get bankCharge() {
  let result = 4.5;
  if (this._daysOverdrawn > 0)
    result += this.type.overdraftCharge(this.daysOverdrawn);
  return result;
}
```

In the earlier steps, I passed `daysOverdrawn` as a parameter—but if there's a lot of data from the account to pass, I might prefer to pass the account itself.

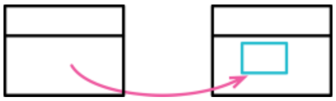
*class Account...*

```
get bankCharge() {
  let result = 4.5;
  if (this._daysOverdrawn > 0) result += this.overdraftCharge;
  return result;
}

get overdraftCharge() {
  return this.type.overdraftCharge(this);
}
```

*class AccountType...*

```
overdraftCharge(account) {
  if (this.isPremium) {
    const baseCharge = 10;
    if (account.daysOverdrawn <= 7)
      return baseCharge;
    else
      return baseCharge + (account.daysOverdrawn - 7) * 0.85;
  }
  else
    return account.daysOverdrawn * 1.75;
}
```

## Move Field



```
class Customer {
  get plan() {return this._plan;}
  get discountRate() {return this._discountRate;}
```

⇓

```
class Customer {
  get plan() {return this._plan;}
  get discountRate() {return this.plan.discountRate;}
```

### Motivation

Programming involves writing a lot of code that implements behavior—but the strength of a program is really founded on its data structures. If I have a good set of data structures that match the problem, then my behavior code is simple and straightforward. But poor data structures lead to lots of code whose job is merely dealing with the poor data. And it's not just messier code that's harder to understand; it also means the data structures obscure what the program is doing.

So, data structures are important—but like most aspects of programming they are hard to get right. I do make an initial analysis to figure out the best data structures, and I've found that experience and techniques like domain-driven design have improved my ability to do that. But despite all my skill and experience, I still find that I frequently make mistakes in that initial design. In the process of programming, I learn more about the problem domain and my data structures. A design decision that is reasonable and correct one week can become wrong in another.

As soon as I realize that a data structure isn't right, it's vital to change it. If I leave my data structures with their blemishes, those blemishes will confuse my thinking and complicate my code far into the future.

I may seek to move data because I find I always need to pass a field from one record whenever I pass another record to a function. Pieces of data that are always passed to functions together are usually best put in a single record in order to clarify their relationship. Change is also a factor; if a change in one record causes a field in another record to change too, that's a sign of a field in the wrong place. If I have to update the same field in multiple structures, that's a sign that it should move to another place where it only needs to be updated once.

I usually do Move Field in the context of a broader set of changes. Once I've moved a field, I find that many of the users of the field are better off accessing that data through the target object rather than the original source. I then change these with later refactorings. Similarly, I may find that I can't do Move Field at the moment due to the way the data is used. I need to refactor some usage patterns first, then do the move.

In my description so far, I'm saying "record," but all this is true of classes and objects too. A class is a record type with attached functions—and these need to be kept healthy just as much as any other data. The attached functions do make it easier to move data around, since the data is encapsulated behind accessor methods. I can move the data, change the accessors, and clients of the accessors will still work. So, this is a refactoring that's easier to do if you have classes, and my description below makes that assumption. If I'm using bare records that don't support encapsulation, I can still make a change like this, but it is more tricky.

## Mechanics

- Ensure the source field is encapsulated.
- Test.
- Create a field (and accessors) in the target.
- Run static checks.
- Ensure there is a reference from the source object to the target object.

  An existing field or method may give you the target. If not, see if you can easily create a method that will do so. Failing that, you may need to create a new field in the source object that can store the target. This may be a permanent change, but you can also do it temporarily until you have done enough refactoring in the broader context.

- Adjust accessors to use the target field.

  If the target is shared between source objects, consider first updating the setter to modify both target and source fields, followed by *Introduce Assertion (302)* to detect inconsistent updates. Once you determine all is well, finish changing the accessors to use the target field.

- Test.

- Remove the source field.
- Test.

## Example

I'm starting here with this customer and contract:

*class Customer…*

```
constructor(name, discountRate) {
  this._name = name;
  this._discountRate = discountRate;
  this._contract = new CustomerContract(dateToday());
}
get discountRate() {return this._discountRate;}
becomePreferred() {
  this._discountRate += 0.03;
  // other nice things
}
applyDiscount(amount) {
  return amount.subtract(amount.multiply(this._discountRate));
}
```

*class CustomerContract…*

```
constructor(startDate) {
  this._startDate = startDate;
}
```

I want to move the discount rate field from the customer to the customer contract.

The first thing I need to use is *Encapsulate Variable (132)* to encapsulate access to the discount rate field.

*class Customer…*

```
constructor(name, discountRate) {
  this._name = name;
  this._setDiscountRate(discountRate);
  this._contract = new CustomerContract(dateToday());
}
get discountRate() {return this._discountRate;}
_setDiscountRate(aNumber) {this._discountRate = aNumber;}
becomePreferred() {
  this._setDiscountRate(this.discountRate + 0.03);
  // other nice things
}
applyDiscount(amount) {
  return amount.subtract(amount.multiply(this.discountRate));
}
```

I use a method to update the discount rate, rather than a property setter, as I don't want to make a public setter for the discount rate.

I add a field and accessors to the customer contract.

*class CustomerContract...*

```
constructor(startDate, discountRate) {
  this._startDate = startDate;
  this._discountRate = discountRate;
}
get discountRate()    {return this._discountRate;}
set discountRate(arg) {this._discountRate = arg;}
```

I now modify the accessors on customer to use the new field. When I did that, I got an error: "Cannot set property 'discountRate' of undefined". This was because `_setDiscountRate` was called before I created the contract object in the constructor. To fix that, I first reverted to the previous state, then used *Slide Statements (223)* to move the `_setDiscountRate` after creating the contract.

*class Customer...*

```
constructor(name, discountRate) {
  this._name = name;
  this._setDiscountRate(discountRate);
  this._contract = new CustomerContract(dateToday());
}
```

I tested that, then changed the accessors again to use the contract.

*class Customer...*

```
get discountRate() {return this._contract.discountRate;}
_setDiscountRate(aNumber) {this._contract.discountRate = aNumber;}
```

Since I'm using JavaScript, there is no declared source field, so I don't need to remove anything further.

## *Changing a Bare Record*

This refactoring is generally easier with objects, since encapsulation provides a natural way to wrap data access in methods. If I have many functions accessing a bare record, then, while it's still a valuable refactoring, it is decidedly more tricky.

I can create accessor functions and modify all the reads and writes to use them. If the field that's being moved is immutable, I can update both the source and the target fields when I set its value and gradually migrate reads. Still, if possible, my first move would be to use *Encapsulate Record (162)* to turn the record into a class so I can make the change more easily.

## Example: Moving to a Shared Object

Now, let's consider a different case. Here's an account with an interest rate:

*class Account…*

```
constructor(number, type, interestRate) {
  this._number = number;
  this._type = type;
  this._interestRate = interestRate;
}
get interestRate() {return this._interestRate;}
```

*class AccountType…*

```
constructor(nameString) {
  this._name = nameString;
}
```

I want to change things so that an account's interest rate is determined from its account type.

The access to the interest rate is already nicely encapsulated, so I'll just create the field and an appropriate accessor on the account type.

*class AccountType…*

```
constructor(nameString, interestRate) {
  this._name = nameString;
  this._interestRate = interestRate;
}
get interestRate() {return this._interestRate;}
```

But there is a potential problem when I update the accesses from account. Before this refactoring, each account had its own interest rate. Now, I want all accounts to share the interest rates of their account type. If all the accounts of the same type already have the same interest rate, then there's no change in observable behavior, so I'm fine with the refactoring. But if there's an account with a different interest rate, it's no longer a refactoring. If my account data is held in a database, I should check the database to ensure that all my accounts have the rate matching their type. I can also *Introduce Assertion (302)* in the account class.

*class Account…*

```
constructor(number, type, interestRate) {
  this._number = number;
  this._type = type;
  assert(interestRate === this._type.interestRate);
  this._interestRate = interestRate;
}
get interestRate() {return this._interestRate;}
```

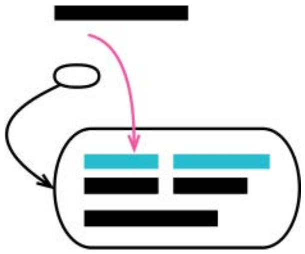
I might run the system for a while with this assertion in place to see if I get an error. Or, instead of adding an assertion, I might log the problem. Once I'm confident that I'm not introducing an observable change, I can change the access, removing the update from the account completely.

*class Account…*

```
constructor(number, type) {
  this._number = number;
  this._type = type;
}
get interestRate() {return this._type.interestRate;}
```

## Move Statements into Function

inverse of: *Move Statements to Callers (217)*



```
result.push(`<p>title: ${person.photo.title}</p>`);
result.concat(photoData(person.photo));

function photoData(aPhoto) {
  return [
    `<p>location: ${aPhoto.location}</p>`,
    `<p>date: ${aPhoto.date.toDateString()}</p>`,
  ];
}
```

⇓

```
result.concat(photoData(person.photo));

function photoData(aPhoto) {
  return [
    `<p>title: ${aPhoto.title}</p>`,
    `<p>location: ${aPhoto.location}</p>`,
    `<p>date: ${aPhoto.date.toDateString()}</p>`,
  ];
}
```

### Motivation

Removing duplication is one of the best rules of thumb of healthy code. If I see the same code executed every time I call a particular function, I look to combine that repeating code into the function itself. That way, any future modifications to the repeating code can be done in one place and used by all the callers. Should the code vary in the future, I can easily move it (or some of it) out again with *Move Statements to Callers (217)*.

I move statements into a function when I can best understand these statements as part of the called function. If they don't make sense as part of the called function, but still should be called with it, I'll simply use *Extract Function (106)* on the statements and the called function. That's essentially the same process as I describe below, but without the inline and rename steps. It's not unusual to do that and then, after later reflection, carry out those final steps.

## Mechanics

- If the repetitive code isn't adjacent to the call of the target function, use *Slide Statements (223)* to get it adjacent.
- If the target function is only called by the source function, just cut the code from the source, paste it into the target, test, and ignore the rest of these mechanics.
- If you have more callers, use *Extract Function (106)* on one of the call sites to extract both the call to the target function and the statements you wish to move into it. Give it a name that's transient, but easy to grep.
- Convert every other call to use the new function. Test after each conversion.
- When all the original calls use the new function, use *Inline Function (115)* to inline the original function completely into the new function, removing the original function.
- *Rename Function (124)* to change the name of the new function to the same name as the original function.

  Or to a better name, if there is one.

## Example

I'll start with this code to emit HTML for data about a photo:

```
function renderPerson(outStream, person) {
  const result = [];
  result.push(`<p>${person.name}</p>`);
  result.push(renderPhoto(person.photo));
  result.push(`<p>title: ${person.photo.title}</p>`);
  result.push(emitPhotoData(person.photo));
  return result.join("\n");
}
```

```
function photoDiv(p) {
  return [
    "<div>",
    `<p>title: ${p.title}</p>`,
    emitPhotoData(p),
    "</div>",
  ].join("\n");
}

function emitPhotoData(aPhoto) {
  const result = [];
  result.push(`<p>location: ${aPhoto.location}</p>`);
  result.push(`<p>date: ${aPhoto.date.toDateString()}</p>`);
  return result.join("\n");
}
```

This code shows two calls to `emitPhotoData`, each preceded by a line of code that is semantically equivalent. I'd like to remove this duplication by moving the title printing into `emitPhotoData`. If I had just the one caller, I would just cut and paste the code, but the more callers I have, the more I'm inclined to use a safer procedure.

I begin by using *Extract Function (106)* on one of the callers. I'm extracting the statements I want to move into `emitPhotoData`, together with the call to `emitPhotoData` itself.

```
function photoDiv(p) {
  return [
    "<div>",
    zznew(p),
    "</div>",
  ].join("\n");
}

function zznew(p) {
  return [
    `<p>title: ${p.title}</p>`,
    emitPhotoData(p),
  ].join("\n");
}
```

I can now look at the other callers of `emitPhotoData` and, one by one, replace the calls and the preceding statements with calls to the new function.

```
function renderPerson(outStream, person) {
  const result = [];
  result.push(`<p>${person.name}</p>`);
  result.push(renderPhoto(person.photo));
  result.push(zznew(person.photo));
  return result.join("\n");
}
```

Now that I've done all the callers, I use *Inline Function (115)* on `emitPhotoData`:

```
function zznew(p) {
  return [
    `<p>title: ${p.title}</p>`,
    `<p>location: ${p.location}</p>`,
    `<p>date: ${p.date.toDateString()}</p>`,
  ].join("\n");
}
```

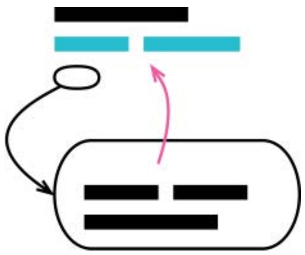
and finish with *Rename Function (124)*:

```
function renderPerson(outStream, person) {
  const result = [];
  result.push(`<p>${person.name}</p>`);
  result.push(renderPhoto(person.photo));
  result.push(emitPhotoData(person.photo));
  return result.join("\n");
}

function photoDiv(aPhoto) {
  return [
    "<div>",
    emitPhotoData(aPhoto),
    "</div>",
  ].join("\n");
}

function emitPhotoData(aPhoto) {
  return [
    `<p>title: ${aPhoto.title}</p>`,
    `<p>location: ${aPhoto.location}</p>`,
    `<p>date: ${aPhoto.date.toDateString()}</p>`,
  ].join("\n");
}
```

I also make the parameter names fit my convention while I'm at it.

## Move Statements to Callers

inverse of: *Move Statements into Function (213)*



```
emitPhotoData(outStream, person.photo);

function emitPhotoData(outStream, photo) {
  outStream.write(`<p>title: ${photo.title}</p>\n`);
  outStream.write(`<p>location: ${photo.location}</p>\n`);
}
```

⇓

```
emitPhotoData(outStream, person.photo);
outStream.write(`<p>location: ${person.photo.location}</p>\n`);

function emitPhotoData(outStream, photo) {
  outStream.write(`<p>title: ${photo.title}</p>\n`);
}
```

### Motivation

Functions are the basic building block of the abstractions we build as programmers. And, as with any abstraction, we don't always get the boundaries right. As a code base changes its capabilities—as most useful software does—we often find our abstraction boundaries shift. For functions, that means that what might once have been a cohesive, atomic unit of behavior becomes a mix of two or more different things.

One trigger for this is when common behavior used in several places needs to vary in some of its calls. Now, we need to move the varying behavior out of the function to its callers. In this case, I'll use *Slide Statements (223)* to get the varying behavior to the beginning or end of the function and then Move Statements to Callers. Once the varying code is in the caller, I can change it when necessary.

Move Statements to Callers works well for small changes, but sometimes the boundaries between caller and callee need complete reworking. In that case, my best move is to use *Inline Function (115)* and then slide and extract new functions to form better boundaries.

## Mechanics

- In simple circumstances, where you have only one or two callers and a simple function to call from, just cut the first line from the called function and paste (and perhaps fit) it into the callers. Test and you're done.
- Otherwise, apply *Extract Function (106)* to all the statements that you *don't* wish to move; give it a temporary but easily searchable name.

  If the function is a method that is overridden by subclasses, do the extraction on all of them so that the remaining method is identical in all classes. Then remove the subclass methods.
- Use *Inline Function (115)* on the original function.
- Apply *Change Function Declaration (124)* on the extracted function to rename it to the original name.

  Or to a better name, if you can think of one.

## Example

Here's a simple case: a function with two callers.

```
function renderPerson(outStream, person) {
  outStream.write(`<p>${person.name}</p>\n`);
  renderPhoto(outStream, person.photo);
  emitPhotoData(outStream, person.photo);
}

function listRecentPhotos(outStream, photos) {
  photos
    .filter(p => p.date > recentDateCutoff())
    .forEach(p => {
      outStream.write("<div>\n");
      emitPhotoData(outStream, p);
      outStream.write("</div>\n");
    });
}

function emitPhotoData(outStream, photo) {
  outStream.write(`<p>title: ${photo.title}</p>\n`);
  outStream.write(`<p>date: ${photo.date.toDateString()}</p>\n`);
  outStream.write(`<p>location: ${photo.location}</p>\n`);
}
```

I need to modify the software so that `listRecentPhotos` renders the location information differently while `renderPerson` stays the same. To make this change easier, I'll use Move Statements to Callers on the final line.

Usually, when faced with something this simple, I'll just cut the last line from `renderPerson` and paste it below the two calls. But since I'm explaining what to do in more tricky cases, I'll go through the more elaborate but safer procedure.

My first step is to use *Extract Function (106)* on the code that will remain in `emitPhotoData`.

```
function renderPerson(outStream, person) {
  outStream.write(`<p>${person.name}</p>\n`);
  renderPhoto(outStream, person.photo);
  emitPhotoData(outStream, person.photo);
}

function listRecentPhotos(outStream, photos) {
  photos
    .filter(p => p.date > recentDateCutoff())
    .forEach(p => {
      outStream.write("<div>\n");
      emitPhotoData(outStream, p);
      outStream.write("</div>\n");
    });
}

function emitPhotoData(outStream, photo) {
  zztmp(outStream, photo);
  outStream.write(`<p>location: ${photo.location}</p>\n`);
}

function zztmp(outStream, photo) {
  outStream.write(`<p>title: ${photo.title}</p>\n`);
  outStream.write(`<p>date: ${photo.date.toDateString()}</p>\n`);
}
```

Usually, the name of the extracted function is only temporary, so I don't worry about coming up with anything meaningful. However, it is helpful to use something that's easy to grep. I can test at this point to ensure the code works over the function call boundary.

Now I use *Inline Function (115)*, one call at a time. I start with `renderPerson`.

```
function renderPerson(outStream, person) {
  outStream.write(`<p>${person.name}</p>\n`);
  renderPhoto(outStream, person.photo);
  zztmp(outStream, person.photo);
  outStream.write(`<p>location: ${person.photo.location}</p>\n`);
}
```

```
function listRecentPhotos(outStream, photos) {
  photos
    .filter(p => p.date > recentDateCutoff())
    .forEach(p => {
      outStream.write("<div>\n");
      emitPhotoData(outStream, p);
      outStream.write("</div>\n");
    });
}

function emitPhotoData(outStream, photo) {
  zztmp(outStream, photo);
  outStream.write(`<p>location: ${photo.location}</p>\n`);
}

function zztmp(outStream, photo) {
  outStream.write(`<p>title: ${photo.title}</p>\n`);
  outStream.write(`<p>date: ${photo.date.toDateString()}</p>\n`);
}
```

I test again to ensure this call is working properly, then move onto the next.

```
function renderPerson(outStream, person) {
  outStream.write(`<p>${person.name}</p>\n`);
  renderPhoto(outStream, person.photo);
  zztmp(outStream, person.photo);
  outStream.write(`<p>location: ${person.photo.location}</p>\n`);
}

function listRecentPhotos(outStream, photos) {
  photos
    .filter(p => p.date > recentDateCutoff())
    .forEach(p => {
      outStream.write("<div>\n");
      zztmp(outStream, p);
      outStream.write(`<p>location: ${p.location}</p>\n`);
      outStream.write("</div>\n");
    });
}

function emitPhotoData(outStream, photo) {
  zztmp(outStream, photo);
  outStream.write(`<p>location: ${photo.location}</p>\n`);
}

function zztmp(outStream, photo) {
  outStream.write(`<p>title: ${photo.title}</p>\n`);
  outStream.write(`<p>date: ${photo.date.toDateString()}</p>\n`);
}
```

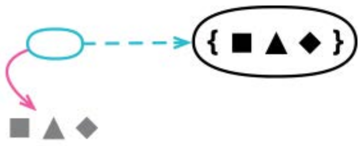
Then I can delete the outer function, completing *Inline Function (115)*.

```
function renderPerson(outStream, person) {
  outStream.write(`<p>${person.name}</p>\n`);
  renderPhoto(outStream, person.photo);
  zztmp(outStream, person.photo);
  outStream.write(`<p>location: ${person.photo.location}</p>\n`);
}

function listRecentPhotos(outStream, photos) {
  photos
    .filter(p => p.date > recentDateCutoff())
    .forEach(p => {
      outStream.write("<div>\n");
      zztmp(outStream, p);
      outStream.write(`<p>location: ${p.location}</p>\n`);
      outStream.write("</div>\n");
    });
}

~~function emitPhotoData(outStream, photo) {~~
~~  zztmp(outStream, photo);~~
~~  outStream.write(`<p>location: ${photo.location}</p>\n`);~~
~~}~~

function zztmp(outStream, photo) {
  outStream.write(`<p>title: ${photo.title}</p>\n`);
  outStream.write(`<p>date: ${photo.date.toDateString()}</p>\n`);
}
```

I then rename `zztmp` back to the original name.

```
function renderPerson(outStream, person) {
  outStream.write(`<p>${person.name}</p>\n`);
  renderPhoto(outStream, person.photo);
  emitPhotoData(outStream, person.photo);
  outStream.write(`<p>location: ${person.photo.location}</p>\n`);
}

function listRecentPhotos(outStream, photos) {
  photos
    .filter(p => p.date > recentDateCutoff())
    .forEach(p => {
      outStream.write("<div>\n");
      emitPhotoData(outStream, p);
      outStream.write(`<p>location: ${p.location}</p>\n`);
      outStream.write("</div>\n");
    });
}

function emitPhotoData(outStream, photo) {
  outStream.write(`<p>title: ${photo.title}</p>\n`);
  outStream.write(`<p>date: ${photo.date.toDateString()}</p>\n`);
}
```

## Replace Inline Code with Function Call



```
let appliesToMass = false;
for(const s of states) {
  if (s === "MA") appliesToMass = true;
}
```

⇓

```
appliesToMass = states.includes("MA");
```

### Motivation

Functions allow me to package up bits of behavior. This is useful for understanding—a named function can explain the purpose of the code rather than its mechanics. It's also valuable to remove duplication: Instead of writing the same code twice, I just call the function. Then, should I need to change the function's implementation, I don't have to track down similar-looking code to update all the changes. (I may have to look at the callers, to see if they should all use the new code, but that's both less common and much easier.)

If I see inline code that's doing the same thing that I have in an existing function, I'll usually want to replace that inline code with a function call. The exception is if I consider the similarity to be coincidental—so that, if I change the function body, I don't expect the behavior in this inline code to change. A guide to this is the name of the function. A good name should make sense in place of inline code I have. If the name doesn't make sense, that may be because it's a poor name (in which case I use *Rename Function (124)* to fix it) or because the function's purpose is different to what I want in this case—so I shouldn't call it.

I find it particularly satisfying to do this with calls to library functions—that way, I don't even have to write the function body.

### Mechanics

- Replace the inline code with a call to the existing function.
- Test.

## Slide Statements

formerly: *Consolidate Duplicate Conditional Fragments*

```
const pricingPlan = retrievePricingPlan();
const order = retreiveOrder();
let charge;
const chargePerUnit = pricingPlan.unit;
```

⇓

```
const pricingPlan = retrievePricingPlan();
const chargePerUnit = pricingPlan.unit;
const order = retreiveOrder();
let charge;
```

### Motivation

Code is easier to understand when things that are related to each other appear together. If several lines of code access the same data structure, it's best for them to be together rather than intermingled with code accessing other data structures. At its simplest, I use Slide Statements to keep such code together. A very common case of this is declaring and using variables. Some people like to declare all their variables at the top of a function. I prefer to declare the variable just before I first use it.

Usually, I move related code together as a preparatory step for another refactoring, often an *Extract Function (106)*. Putting related code into a clearly separated function is a better separation than just moving a set of lines together, but I can't do the *Extract Function (106)* unless the code is together in the first place.

### Mechanics

- Identify the target position to move the fragment to. Examine statements between source and target to see if there is interference for the candidate fragment. Abandon action if there is any interference.

  A fragment cannot slide backwards earlier than any element it references is declared.

A fragment cannot slide forwards beyond any element that references it.

A fragment cannot slide over any statement that modifies an element it references.

A fragment that modifies an element cannot slide over any other element that references the modified element.

- Cut the fragment from the source and paste into the target position.
- Test.

If the test fails, try breaking down the slide into smaller steps. Either slide over less code or reduce the amount of code in the fragment you're moving.

## Example

When sliding code fragments, there are two decisions involved: what slide I'd like to do and whether I can do it. The first decision is very context-specific. On the simplest level, I like to declare elements close to where I use them, so I'll often slide a declaration down to its usage. But almost always I slide some code because I want to do another refactoring—perhaps to get a clump of code together to *Extract Function (106)*.

Once I have a sense of where I'd like to move some code, the next part is deciding if I can do it. This involves looking at the code I'm sliding and the code I'm sliding over: Do they interfere with each other in a way that would change the observable behavior of the program?

Consider the following fragment of code:

```
 1 const pricingPlan = retrievePricingPlan();
 2 const order = retreiveOrder();
 3 const baseCharge = pricingPlan.base;
 4 let charge;
 5 const chargePerUnit = pricingPlan.unit;
 6 const units = order.units;
 7 let discount;
 8 charge = baseCharge + units * chargePerUnit;
 9 let discountableUnits = Math.max(units - pricingPlan.discountThreshold, 0);
10 discount = discountableUnits * pricingPlan.discountFactor;
11 if (order.isRepeat) discount += 20;
12 charge = charge - discount;
13 chargeOrder(charge);
```

The first seven lines are declarations, and it's relatively easy to move these. For example, I may want to move all the code dealing with discounts together, which would involve moving line 7 (`` `let discount` ``) to above line 10 (`` `discount = ...` ``). Since a declaration has no side effects and refers to no other variable, I can safely move this forwards as far as the first line that references `discount` itself. This is also a common move—if I want to use *Extract Function (106)* on the discount logic, I'll need to move the declaration down first.

I do similar analysis with any code that doesn't have side effects. So I can take line 2 (`` `const order = ...` ``) and move it down to above line 6 (`` `const units = ...` ``) without trouble.

In this case, I'm also helped by the fact that the code I'm moving over doesn't have side effects either. Indeed, I can freely rearrange code that lacks side effects to my heart's content, which is one of the reasons why wise programmers prefer to use side-effect-free code as much as possible.

There is a wrinkle here, however. How do I know that line 2 is side-effect-free? To be sure, I'd need to look inside `retrieveOrder()` to ensure there are no side effects there (and inside any functions it calls, and inside any functions its functions call, and so on). In practice, when working on my own code, I know that I generally follow the Command-Query Separation [mf-cqs] principle, so any function that returns a value is free of side effects. But I can only be confident of that because I know the code base; if I were working in an unknown code base, I'd have to be more cautious. But I do try to follow the Command-Query Separation in my own code because it's so valuable to know that code is free of side effects.

When sliding code that has a side effect, or sliding over code with side effects, I have to be much more careful. What I'm looking for is interference between the two code fragments. So, let's say I want to slide line 11 (`` `if (order.isRepeat) ...` ``) down to the end. I'm prevented from doing that by line 12 because it references the variable whose state I'm changing in line 11. Similarly, I can't take line 13 (`` `chargeOrder(charge)` ``) and move it up because line 12 modifies some state that line 13 references. However, I can slide line 8 (`` `charge = baseCharge + ...` ``) over lines 9–11 because there they don't modify any common state.

The most straightforward rule to follow is that I can't slide one fragment of code over another if any data that both fragments refer to is modified by either one. But that's not a comprehensive rule; I can happily slide either of the following two lines over the other:

```
a = a + 10;
a = a + 5;
```

But judging whether a slide is safe means I have to really understand the operations involved and how they compose.

Since I need to worry so much about updating state, I look to remove as much of it as I can. So with this code, I'd be looking to apply *Split Variable (240)* on `charge` before I indulge in any sliding around of that code.

Here, the analysis is relatively simple because I'm mostly just modifying local variables. With more complex data structures, it's much harder to be sure when I get interference. So tests play an important role: Slide the fragment, run tests, see if things break. If my test coverage is good, I can feel happy with the refactoring. But if tests aren't reliable, I need to be more wary—or, more likely, to improve the tests for the code I'm working on.

The most important consequence of a test failure after a slide is to use smaller slides: Instead of sliding over ten lines, I'll just pick five, or slide up to what I reckon is a dangerous line. It may also mean that the slide isn't worth it, and I need to work on something else first.

## Example: Sliding with Conditionals

I can also do slides with conditionals. This will either involve removing duplicate logic when I slide out of a conditional, or adding duplicate logic when I slide in.

Here's a case where I have the same statements in both legs of a conditional:

```
let result;
if (availableResources.length === 0) {
  result = createResource();
  allocatedResources.push(result);
} else {
  result = availableResources.pop();
  allocatedResources.push(result);
}
return result;
```

I can slide these out of the conditional, in which case they turn into a single statement outside of the conditional block.

```
let result;
if (availableResources.length === 0) {
  result = createResource();
} else {
  result = availableResources.pop();
}
allocatedResources.push(result);
return result;
```

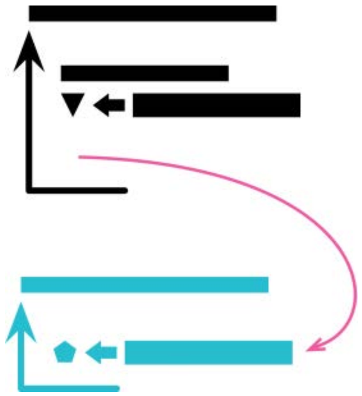
In the reverse case, sliding a fragment into a conditional means repeating it in every leg of the conditional.

## Further Reading

I've seen an almost identical refactoring under the name of Swap Statement [wake-swap]. Swap Statement moves adjacent fragments, but it only works with single-statement fragments. You can think of it as Slide Statements where both the sliding fragment and the slid-over fragment are single statements. This refactoring appeals to me; after all, I'm always going on about taking small steps—steps that may seem ridiculously small to those new to refactoring.

But I ended up writing this refactoring with larger fragments because that is what I do. I only move one statement at a time if I'm having difficulty with a larger slide, and I rarely run into problems with larger slides. With more messy code, however, smaller slides end up being easier.

## Split Loop



```
let averageAge = 0;
let totalSalary = 0;
for (const p of people) {
  averageAge += p.age;
  totalSalary += p.salary;
}
averageAge = averageAge / people.length;
```

$\Downarrow$

```
let totalSalary = 0;
for (const p of people) {
  totalSalary += p.salary;
}

let averageAge = 0;
for (const p of people) {
  averageAge += p.age;
}
averageAge = averageAge / people.length;
```

### Motivation

You often see loops that are doing two different things at once just because they can do that with one pass through a loop. But if you're doing two different things in the same loop, then whenever you need to modify the loop you have to understand both things. By splitting the loop, you ensure you only need to understand the behavior you need to modify.

Splitting a loop can also make it easier to use. A loop that calculates a single value can just return that value. Loops that do many things need to return structures or populate local variables. I frequently follow a sequence of Split Loop followed by *Extract Function (106)*.

Many programmers are uncomfortable with this refactoring, as it forces you to execute the loop twice. My reminder, as usual, is to separate refactoring from optimization (*Refactoring and Performance (64)*). Once I have my code clear, I'll optimize it, and if the loop traversal is a bottleneck, it's easy to slam the loops back together. But the actual iteration through even a large list is rarely a bottleneck, and splitting the loops often enables other, more powerful, optimizations.

## Mechanics

- Copy the loop.
- Identify and eliminate duplicate side effects.
- Test.

When done, consider *Extract Function (106)* on each loop.

## Example

I'll start with a little bit of code that calculates the total salary and youngest age.

```
let youngest = people[0] ? people[0].age : Infinity;
let totalSalary = 0;
for (const p of people) {
  if (p.age < youngest) youngest = p.age;
  totalSalary += p.salary;
}

return `youngestAge: ${youngest}, totalSalary: ${totalSalary}`;
```

It's a very simple loop, but it's doing two different calculations. To split them, I begin with just copying the loop.

```
let youngest = people[0] ? people[0].age : Infinity;
let totalSalary = 0;
for (const p of people) {
  if (p.age < youngest) youngest = p.age;
  totalSalary += p.salary;
}
```

```
  for (const p of people) {
    if (p.age < youngest) youngest = p.age;
    totalSalary += p.salary;
  }

  return `youngestAge: ${youngest}, totalSalary: ${totalSalary}`;
```

With the loop copied, I need to remove the duplication that would otherwise produce wrong results. If something in the loop has no side effects, I can leave it there for now, but it's not the case with this example.

```
let youngest = people[0] ? people[0].age : Infinity;
let totalSalary = 0;
for (const p of people) {
  ~~if (p.age < youngest) youngest = p.age;~~
  totalSalary += p.salary;
}

for (const p of people) {
  if (p.age < youngest) youngest = p.age;
  ~~totalSalary += p.salary;~~
}

return `youngestAge: ${youngest}, totalSalary: ${totalSalary}`;
```

Officially, that's the end of the Split Loop refactoring. But the point of Split Loop isn't what it does on its own but what it sets up for the next move—and I'm usually looking to extract the loops into their own functions. I'll use *Slide Statements (223)* to reorganize the code a bit first.

```
let totalSalary = 0;
for (const p of people) {
  totalSalary += p.salary;
}

let youngest = people[0] ? people[0].age : Infinity;
for (const p of people) {
  if (p.age < youngest) youngest = p.age;
}

return `youngestAge: ${youngest}, totalSalary: ${totalSalary}`;
```

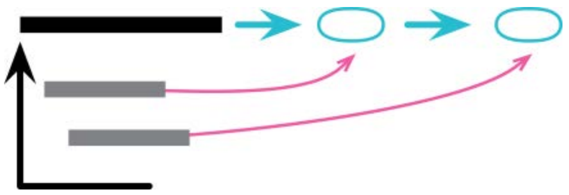
Then I do a couple of *Extract Function (106)*.

```
return `youngestAge: ${youngestAge()}, totalSalary: ${totalSalary()}`;

function totalSalary() {
  let totalSalary = 0;
  for (const p of people) {
    totalSalary += p.salary;
  }
  return totalSalary;
}

function youngestAge() {
  let youngest = people[0] ? people[0].age : Infinity;
  for (const p of people) {
    if (p.age < youngest) youngest = p.age;
  }
  return youngest;
}
```

I can rarely resist *Replace Loop with Pipeline (231)* for the total salary, and there's an obvious *Substitute Algorithm (195)* for the youngest age.

```
return `youngestAge: ${youngestAge()}, totalSalary: ${totalSalary()}`;

function totalSalary() {
  return people.reduce((total,p) => total + p.salary, 0);
}
function youngestAge() {
  return Math.min(...people.map(p => p.age));
}
```

## Replace Loop with Pipeline



```
const names = [];
for (const i of input) {
  if (i.job === "programmer")
    names.push(i.name);
}
```

⇓

```
const names = input
  .filter(i => i.job === "programmer")
  .map(i => i.name)
;
```

### Motivation

Like most programmers, I was taught to use loops to iterate over a collection of objects. Increasingly, however, language environments provide a better construct: the collection pipeline. Collection Pipelines [mf-cp] allow me to describe my processing as a series of operations, each consuming and emitting a collection. The most common of these operations are *map*, which uses a function to transform each element of the input collection, and *filter* which uses a function to select a subset of the input collection for later steps in the pipeline. I find logic much easier to follow if it is expressed as a pipeline—I can then read from top to bottom to see how objects flow through the pipeline.

### Mechanics

- Create a new variable for the loop's collection.

  This may be a simple copy of an existing variable.

- Starting at the top, take each bit of behavior in the loop and replace it with a collection pipeline operation in the derivation of the loop collection variable. Test after each change.
- Once all behavior is removed from the loop, remove it.

  If it assigns to an accumulator, assign the pipeline result to the accumulator.

## Example

I'll begin with some data: a CSV file of data about our offices.

```
office, country, telephone
Chicago, USA, +1 312 373 1000
Beijing, China, +86 4008 900 505
Bangalore, India, +91 80 4064 9570
Porto Alegre, Brazil, +55 51 3079 3550
Chennai, India, +91 44 660 44766

... (more data follows)
```

The following function picks out the offices in India and returns their cities and telephone numbers:

```
function acquireData(input) {
  const lines = input.split("\n");
  let firstLine = true;
  const result = [];
  for (const line of lines) {
    if (firstLine) {
      firstLine = false;
      continue;
    }
    if (line.trim() === "") continue;
    const record = line.split(",");
    if (record[1].trim() === "India") {
      result.push({city: record[0].trim(), phone: record[2].trim()});
    }
  }
  return result;
}
```

I want to replace that loop with a collection pipeline.

My first step is to create a separate variable for the loop to work over.

```
function acquireData(input) {
  const lines = input.split("\n");
  let firstLine = true;
  const result = [];
  const loopItems = lines
  for (const line of loopItems) {
    if (firstLine) {
      firstLine = false;
      continue;
    }
    if (line.trim() === "") continue;
    const record = line.split(",");
    if (record[1].trim() === "India") {
      result.push({city: record[0].trim(), phone: record[2].trim()});
    }
  }
  return result;
}
```

The first part of the loop is all about skipping the first line of the CSV file. This calls for a slice, so I remove that first section of the loop and add a slice operation to the formation of the loop variable.

```
function acquireData(input) {
  const lines = input.split("\n");
  ~~let firstLine = true;~~
  const result = [];
  const loopItems = lines
        .slice(1);
  for (const line of loopItems) {
    ~~if (firstLine) {~~
      ~~firstLine = false;~~
      ~~continue;~~
    ~~}~~
    if (line.trim() === "") continue;
    const record = line.split(",");
    if (record[1].trim() === "India") {
      result.push({city: record[0].trim(), phone: record[2].trim()});
    }
  }
  return result;
}
```

As a bonus, this lets me delete `firstLine`—and I particularly enjoy deleting control variables.

The next bit of behavior removes any blank lines. I can replace this with a filter operation.

```
function acquireData(input) {
  const lines = input.split("\n");
  const result = [];
  const loopItems = lines
        .slice(1)
        .filter(line => line.trim() !== "")
        ;
  for (const line of loopItems) {
    ~~if (line.trim() === "") continue;~~
    const record = line.split(",");
    if (record[1].trim() === "India") {
      result.push({city: record[0].trim(), phone: record[2].trim()});
    }
  }
  return result;
}
```

When writing a pipeline, I find it best to put the terminal semicolon on its own line.

I use the map operation to turn lines into an array of strings—misleadingly called record in the original function, but it's safer to keep the name for now and rename later.

```
function acquireData(input) {
  const lines = input.split("\n");
  const result = [];
  const loopItems = lines
        .slice(1)
        .filter(line => line.trim() !== "")
        .map(line => line.split(","))
        ;
  for (const line of loopItems) {
    const record = line;~~.split(",");~~
    if (record[1].trim() === "India") {
      result.push({city: record[0].trim(), phone: record[2].trim()});
    }
  }
  return result;
}
```

Filter again to just get the India records:

```
function acquireData(input) {
  const lines = input.split("\n");
  const result = [];
  const loopItems = lines
        .slice(1)
        .filter(line => line.trim() !== "")
        .map(line => line.split(","))
        .filter(record => record[1].trim() === "India")
        ;
  for (const line of loopItems) {
    const record = line;
    ~~if (record[1].trim() === "India") {~~
      result.push({city: record[0].trim(), phone: record[2].trim()});
    ~~}~~
  }
  return result;
}
```

Map to the output record form:

```
function acquireData(input) {
  const lines = input.split("\n");
  const result = [];
  const loopItems = lines
        .slice(1)
        .filter(line => line.trim() !== "")
        .map(line => line.split(","))
        .filter(record => record[1].trim() === "India")
        .map(record => ({city: record[0].trim(), phone: record[2].trim()}))
        ;
  for (const line of loopItems) {
    const record = line;
    result.push(line);
  }
  return result;
}
```

Now, all the loop does is assign values to the accumulator. So I can remove it and assign the result of the pipeline to the accumulator:

```
function acquireData(input) {
  const lines = input.split("\n");
  const result = lines
        .slice(1)
        .filter(line => line.trim() !== "")
        .map(line => line.split(","))
        .filter(record => record[1].trim() === "India")
        .map(record => ({city: record[0].trim(), phone: record[2].trim()}))
        ;
  ~~for (const line of loopItems) {~~
    ~~const record = line;~~
    ~~result.push(line);~~
  ~~}~~
  return result;
}
```

That's the core of the refactoring. But I do have some cleanup I'd like to do. I inlined `result`, renamed some lambda variables, and made the layout read more like a table.

```
function acquireData(input) {
  const lines = input.split("\n");
  return lines
        .slice  (1)
        .filter (line   => line.trim() !== "")
        .map    (line   => line.split(","))
        .filter (fields => fields[1].trim() === "India")
        .map    (fields => ({city: fields[0].trim(), phone: fields[2].trim()}))
        ;
}
```

I thought about inlining `lines` too, but felt that its presence explains what's happening.

## Further Reading

For more examples on turning loops into pipelines, see my essay "Refactoring with Loops and Collection Pipelines" [mf-ref-pipe].

## Remove Dead Code

```
if(false) {
  doSomethingThatUsedToMatter();
}
```

⇓

```

```

### Motivation

When we put code into production, even on people's devices, we aren't charged by weight. A few unused lines of code don't slow down our systems nor take up significant memory; indeed, decent compilers will instinctively remove them. But unused code is still a significant burden when trying to understand how the software works. It doesn't carry any warning signs telling programmers that they can ignore this function as it's never called any more, so they still have to spend time understanding what it's doing and why changing it doesn't seem to alter the output as they expected.

Once code isn't used any more, we should delete it. I don't worry that I may need it sometime in the future; should that happen, I have my version control system so I can always dig it out again. If it's something I really think I may need one day, I might put a comment into the code that mentions the lost code and which revision it was removed in—but, honestly, I can't remember the last time I did that, or regretted that I hadn't done it.

Commenting out dead code was once a common habit. This was useful in the days before version control systems were widely used, or when they were inconvenient. Now, when I can put even the smallest code base under version control, that's no longer needed.

### Mechanics

- If the dead code can be referenced from outside, e.g., when it's a full function, do a search to check for callers.
- Remove the dead code.
- Test.

*This page intentionally left blank*

# Chapter 9

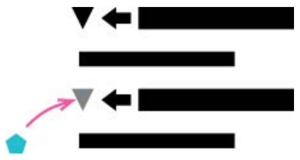
# Organizing Data

Data structures play an important role in our programs, so it's no great shock that I have a clutch of refactorings that focus on them. A value that's used for different purposes is a breeding ground for confusion and bugs—so, when I see one, I use *Split Variable (240)* to separate the usages. As with any program element, getting a variable's name right is tricky and important, so *Rename Variable (137)* is often my friend. But sometimes the best thing I can do with a variable is to get rid of it completely—with *Replace Derived Variable with Query (248)*.

I often find problems in a code base due to a confusion between references and values, so I use *Change Reference to Value (252)* and *Change Value to Reference (256)* to change between these styles.

## Split Variable

formerly: *Remove Assignments to Parameters*
formerly: *Split Temp*



```
let temp = 2 * (height + width);
console.log(temp);
temp = height * width;
console.log(temp);
```

⇓

```
const perimeter = 2 * (height + width);
console.log(perimeter);
const area = height * width;
console.log(area);
```

### Motivation

Variables have various uses. Some of these uses naturally lead to the variable being assigned to several times. Loop variables change for each run of a loop (such as the `i` in `for (let i=0; i<10; i++)`). Collecting variables store a value that is built up during the method.

Many other variables are used to hold the result of a long-winded bit of code for easy reference later. These kinds of variables should be set only once. If they are set more than once, it is a sign that they have more than one responsibility within the method. Any variable with more than one responsibility should be replaced with multiple variables, one for each responsibility. Using a variable for two different things is very confusing for the reader.

### Mechanics

- Change the name of the variable at its declaration and first assignment.

  If the later assignments are of the form `i = i + something`, that is a collecting variable, so don't split it. A collecting variable is often used for calculating sums, string concatenation, writing to a stream, or adding to a collection.

- If possible, declare the new variable as immutable.
- Change all references of the variable up to its second assignment.
- Test.
- Repeat in stages, at each stage renaming the variable at the declaration and changing references until the next assignment, until you reach the final assignment.

## Example

For this example, I compute the distance traveled by a haggis. From a standing start, a haggis experiences an initial force. After a delay, a secondary force kicks in to further accelerate the haggis. Using the common laws of motion, I can compute the distance traveled as follows:

```
function distanceTravelled (scenario, time) {
  let result;
  let acc = scenario.primaryForce / scenario.mass;
  let primaryTime = Math.min(time, scenario.delay);
  result = 0.5 * acc * primaryTime * primaryTime;
  let secondaryTime = time - scenario.delay;
  if (secondaryTime > 0) {
    let primaryVelocity = acc * scenario.delay;
    acc = (scenario.primaryForce + scenario.secondaryForce) / scenario.mass;
    result += primaryVelocity * secondaryTime + 0.5 * acc * secondaryTime * secondaryTime;
  }
  return result;
}
```

A nice awkward little function. The interesting thing for our example is the way the variable `acc` is set twice. It has two responsibilities: one to hold the initial acceleration from the first force and another later to hold the acceleration from both forces. I want to split this variable.

When trying to understand how a variable is used, it's handy if my editor can highlight all occurrences of a symbol within a function or file. Most modern editors can do this pretty easily.

I start at the beginning by changing the name of the variable and declaring the new name as `const`. Then, I change all references to the variable from that point up to the next assignment. At the next assignment, I declare it:

```
function distanceTravelled (scenario, time) {
  let result;
  const primaryAcceleration = scenario.primaryForce / scenario.mass;
  let primaryTime = Math.min(time, scenario.delay);
  result = 0.5 * primaryAcceleration * primaryTime * primaryTime;
  let secondaryTime = time - scenario.delay;
  if (secondaryTime > 0) {
    let primaryVelocity = primaryAcceleration * scenario.delay;
    let acc = (scenario.primaryForce + scenario.secondaryForce) / scenario.mass;
    result += primaryVelocity * secondaryTime + 0.5 * acc * secondaryTime * secondaryTime;
  }
  return result;
}
```

I choose the new name to represent only the first use of the variable. I make it `const` to ensure it is only assigned once. I can then declare the original variable at its second assignment. Now I can compile and test, and all should work.

I continue on the second assignment of the variable. This removes the original variable name completely, replacing it with a new variable named for the second use.

```
function distanceTravelled (scenario, time) {
  let result;
  const primaryAcceleration = scenario.primaryForce / scenario.mass;
  let primaryTime = Math.min(time, scenario.delay);
  result = 0.5 * primaryAcceleration * primaryTime * primaryTime;
  let secondaryTime = time - scenario.delay;
  if (secondaryTime > 0) {
    let primaryVelocity = primaryAcceleration * scenario.delay;
    const secondaryAcceleration = (scenario.primaryForce + scenario.secondaryForce) / scenario.mass;
    result += primaryVelocity * secondaryTime +
      0.5 * secondaryAcceleration * secondaryTime * secondaryTime;
  }
  return result;
}
```

I'm sure you can think of a lot more refactoring to be done here. Enjoy it. (I'm sure it's better than eating the haggis—do you know what they put in those things?)

## Example: Assigning to an Input Parameter

Another case of splitting a variable is where the variable is declared as an input parameter. Consider something like

```
function discount (inputValue, quantity) {
  if (inputValue > 50) inputValue = inputValue - 2;
  if (quantity > 100) inputValue = inputValue - 1;
  return inputValue;
}
```

Here `inputValue` is used both to supply an input to the function and to hold the result for the caller. (Since JavaScript has call-by-value parameters, any modification of `inputValue` isn't seen by the caller.)

In this situation, I would split that variable.

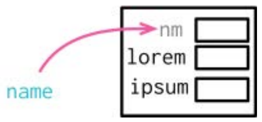
```
function discount (originalInputValue, quantity) {
  let inputValue = originalInputValue;
  if (inputValue > 50) inputValue = inputValue - 2;
  if (quantity > 100) inputValue = inputValue - 1;
  return inputValue;
}
```

I then perform *Rename Variable (137)* twice to get better names.

```
function discount (inputValue, quantity) {
  let result = inputValue;
  if (inputValue > 50) result = result - 2;
  if (quantity > 100) result = result - 1;
  return result;
}
```

You'll notice that I changed the second line to use `inputValue` as its data source. Although the two are the same, I think that line is really about applying the modification to the result value based on the original input value, not the (coincidentally same) value of the result accumulator.

## Rename Field



```
class Organization {
  get name() {...}
}
```

⇓

```
class Organization {
  get title() {...}
}
```

### Motivation

Names are important, and field names in record structures can be especially important when those record structures are widely used across a program. Data structures play a particularly important role in understanding. Many years ago Fred Brooks said, "Show me your flowcharts and conceal your tables, and I shall continue to be mystified. Show me your tables, and I won't usually need your flowcharts; they'll be obvious." While I don't see many people drawing flowcharts these days, the adage remains valid. Data structures are the key to understanding what's going on.

Since these data structures are so important, it's essential to keep them clear. Like anything else, my understanding of data improves the more I work on the software, so it's vital that this improved understanding is embedded into the program.

You may want to rename a field in a record structure, but the idea also applies to classes. Getter and setter methods form an effective field for users of the class. Renaming them is just as important as with bare record structures.

### Mechanics

- If the record has limited scope, rename all accesses to the field and test; no need to do the rest of the mechanics.

- If the record isn't already encapsulated, apply *Encapsulate Record (162)*.
- Rename the private field inside the object, adjust internal methods to fit.
- Test.
- If the constructor uses the name, apply *Change Function Declaration (124)* to rename it.
- Apply *Rename Function (124)* to the accessors.

## Example: Renaming a Field

I'll start with a constant.

```
const organization = {name: "Acme Gooseberries", country: "GB"};
```

I want to change "name" to "title". The object is widely used in the code base, and there are updates to the title in the code. So my first move is to apply *Encapsulate Record (162)*.

```
class Organization {
  constructor(data) {
    this._name = data.name;
    this._country = data.country;
  }
  get name()    {return this._name;}
  set name(aString) {this._name = aString;}
  get country()    {return this._country;}
  set country(aCountryCode) {this._country = aCountryCode;}
}

const organization = new Organization({name: "Acme Gooseberries", country: "GB"});
```

Now that I've encapsulated the record structure into the class, there are four places I need to look at for renaming: the getting function, the setting function, the constructor, and the internal data structure. While that may sound like I've increased my workload, it actually makes my work easier since I can now change these independently instead of all at once, taking smaller steps. Smaller steps mean fewer things to go wrong in each step—therefore, less work. It wouldn't be less work if I never made mistakes—but not making mistakes is a fantasy I gave up on a long time ago.

Since I've copied the input data structure into the internal data structure, I need to separate them so I can work on them independently. I can do this by defining a separate field and adjusting the constructor and accessors to use it.

*class Organization…*

```
class Organization {
  constructor(data) {
    this._title = data.name;
    this._country = data.country;
  }
  get name()    {return this._title;}
  set name(aString) {this._title = aString;}
  get country()    {return this._country;}
  set country(aCountryCode) {this._country = aCountryCode;}
}
```

Next, I add support for using "title" in the constructor.

*class Organization…*

```
class Organization {
  constructor(data) {
    this._title = (data.title !== undefined) ? data.title : data.name;
    this._country = data.country;
  }
  get name()    {return this._title;}
  set name(aString) {this._title = aString;}
  get country()    {return this._country;}
  set country(aCountryCode) {this._country = aCountryCode;}
}
```

Now, callers of my constructor can use either name or title (with title taking precedence). I can now go through all constructor callers and change them one-by-one to use the new name.

```
const organization = new Organization({title: "Acme Gooseberries", country: "GB"});
```

Once I've done all of them, I can remove the support for the name.

*class Organization…*

```
class Organization {
  constructor(data) {
    this._title = data.title;
    this._country = data.country;
  }
  get name()    {return this._title;}
  set name(aString) {this._title = aString;}
  get country()    {return this._country;}
  set country(aCountryCode) {this._country = aCountryCode;}
}
```

Now that the constructor and data use the new name, I can change the accessors, which is as simple as applying *Rename Function (124)* to each one.

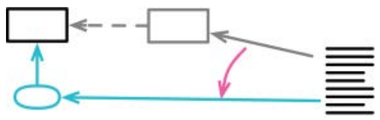
*class Organization…*

```
class Organization {
  constructor(data) {
    this._title = data.title;
    this._country = data.country;
  }
  get title()    {return this._title;}
  set title(aString) {this._title = aString;}
  get country()    {return this._country;}
  set country(aCountryCode) {this._country = aCountryCode;}
}
```

I've shown this process in its most heavyweight form needed for a widely used data structure. If it's being used only locally, as in a single function, I can probably just rename the various properties in one go without doing encapsulation. It's a matter of judgment when to apply to the full mechanics here—but, as usual with refactoring, if my tests break, that's a sign I need to use the more gradual procedure.

Some languages allow me to make a data structure immutable. In this case, rather than encapsulating it, I can copy the value to the new name, gradually change the users, then remove the old name. Duplicating data is a recipe for disaster with mutable data structures; removing such disasters is why immutable data is so popular.

## Replace Derived Variable with Query



```
get discountedTotal() {return this._discountedTotal;}
set discount(aNumber) {
  const old = this._discount;
  this._discount = aNumber;
  this._discountedTotal += old - aNumber;
}
```

⇓

```
get discountedTotal() {return this._baseTotal - this._discount;}
set discount(aNumber) {this._discount = aNumber;}
```

### Motivation

One of the biggest sources of problems in software is mutable data. Data changes can often couple together parts of code in awkward ways, with changes in one part leading to knock-on effects that are hard to spot. In many situations it's not realistic to entirely remove mutable data—but I do advocate minimizing the scope of mutable data at much as possible.

One way I can make a big impact is by removing any variables that I could just as easily calculate. A calculation often makes it clearer what the meaning of the data is, and it is protected from being corrupted when you fail to update the variable as the source data changes.

A reasonable exception to this is when the source data for the calculation is immutable and we can force the result to being immutable too. Transformation operations that create new data structures are thus reasonable to keep even if they could be replaced with calculations. Indeed, there is a duality here between objects that wrap a data structure with a series of calculated properties and functions that transform one data structure into another. The object route is clearly better when the source data changes and you would have to manage the lifetime of the derived data structures. But if the source data is immutable, or the derived data is very transient, then both approaches are effective.

## Mechanics

- Identify all points of update for the variable. If necessary, use *Split Variable (240)* to separate each point of update.
- Create a function that calculates the value of the variable.
- Use *Introduce Assertion (302)* to assert that the variable and the calculation give the same result whenever the variable is used.

  If necessary, use *Encapsulate Variable (132)* to provide a home for the assertion.
- Test.
- Replace any reader of the variable with a call to the new function.
- Test.
- Apply *Remove Dead Code (237)* to the declaration and updates to the variable.

## Example

Here's a small but perfectly formed example of ugliness:

*class ProductionPlan…*

```
get production() {return this._production;}
applyAdjustment(anAdjustment) {
  this._adjustments.push(anAdjustment);
  this._production += anAdjustment.amount;
}
```

Ugliness is in the eye of beholder; here, I see ugliness in duplication—not the common duplication of code but duplication of data. When I apply an adjustment, I'm not just storing that adjustment but also using it to modify an accumulator. I can just calculate that value, without having to update it.

But I'm a cautious fellow. It is my hypothesis is that I can just calculate it—I can test that hypothesis by using *Introduce Assertion (302)*:

*class ProductionPlan…*

```
get production() {
  assert(this._production === this.calculatedProduction);
  return this._production;
}

 get calculatedProduction() {
   return this._adjustments
     .reduce((sum, a) => sum + a.amount, 0);
 }
```

With the assertion in place, I run my tests. If the assertion doesn't fail, I can replace returning the field with returning the calculation:

*class ProductionPlan…*

```
get production() {
  ~~assert(this._production === this.calculatedProduction);~~
  return this.calculatedProduction;
}
```

Then *Inline Function (115)*:

*class ProductionPlan…*

```
get production() {
  return this._adjustments
    .reduce((sum, a) => sum + a.amount, 0);
}
```

I clean up any references to the old variable with *Remove Dead Code (237)*:

*class ProductionPlan…*

```
applyAdjustment(anAdjustment) {
  this._adjustments.push(anAdjustment);
  ~~this._production += anAdjustment.amount;~~
}
```

## Example: More Than One Source

The above example is nice and easy because there's clearly a single source for the value of `production`. But sometimes, more than one element can combine in the accumulator.

*class ProductionPlan…*

```
constructor (production) {
  this._production = production;
  this._adjustments = [];
}
get production() {return this._production;}
applyAdjustment(anAdjustment) {
  this._adjustments.push(anAdjustment);
  this._production += anAdjustment.amount;
}
```

If I do the same *Introduce Assertion (302)* that I did above, it will now fail for any case where the initial value of the production isn't zero.

But I can still replace the derived data. The only difference is that I must first apply *Split Variable (240)*.

```
constructor (production) {
  this._initialProduction = production;
  this._productionAccumulator = 0;
  this._adjustments = [];
}
get production() {
  return this._initialProduction + this._productionAccumulator;
}
```

Now I can *Introduce Assertion (302)*:

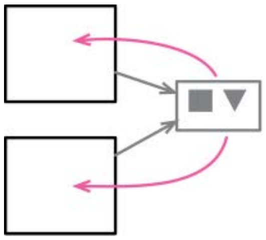
*class ProductionPlan…*

```
get production() {
  assert(this._productionAccumulator === this.calculatedProductionAccumulator);
  return this._initialProduction + this._productionAccumulator;
}

  get calculatedProductionAccumulator() {
    return this._adjustments
      .reduce((sum, a) => sum + a.amount, 0);
  }
```

and continue pretty much as before. I'd be inclined, however, to leave `totalProductionAjustments` as its own property, without inlining it.

## Change Reference to Value

inverse of: *Change Value to Reference (256)*



```
class Product {
  applyDiscount(arg) {this._price.amount -= arg;}
```

⇓

```
class Product {
  applyDiscount(arg) {
    this._price = new Money(this._price.amount - arg, this._price.currency);
  }
```

### Motivation

When I nest an object, or data structure, within another I can treat the inner object as a reference or as a value. The difference is most obviously visible in how I handle updates of the inner object's properties. If I treat it as a reference, I'll update the inner object's property keeping the same inner object. If I treat it as a value, I will replace the entire inner object with a new one that has the desired property.

If I treat a field as a value, I can change the class of the inner object to make it a Value Object [mf-vo]. Value objects are generally easier to reason about, particularly because they are immutable. In general, immutable data structures are easier to deal with. I can pass an immutable data value out to other parts of the program and not worry that it might change without the enclosing object being aware of the change. I can replicate values around my program and not worry about maintaining memory links. Value objects are especially useful in distributed and concurrent systems.

This also suggests when I shouldn't do this refactoring. If I want to share an object between several objects so that any change to the shared object is visible to all its collaborators, then I need the shared object to be a reference.

## Mechanics

- Check that the candidate class is immutable or can become immutable.
- For each setter, apply *Remove Setting Method (331)*.
- Provide a value-based equality method that uses the fields of the value object.

  Most language environments provide an overridable equality function for this purpose. Usually you must override a hashcode generator method as well.

## Example

Imagine we have a person object that holds onto a crude telephone number.

*class Person...*

```
constructor() {
  this._telephoneNumber = new TelephoneNumber();
}

get officeAreaCode()    {return this._telephoneNumber.areaCode;}
set officeAreaCode(arg) {this._telephoneNumber.areaCode = arg;}
get officeNumber()    {return this._telephoneNumber.number;}
set officeNumber(arg) {this._telephoneNumber.number = arg;}
```

*class TelephoneNumber...*

```
get areaCode()    {return this._areaCode;}
set areaCode(arg) {this._areaCode = arg;}

get number()    {return this._number;}
set number(arg) {this._number = arg;}
```

This situation is the result of an *Extract Class (182)* where the old parent still holds update methods for the new object. This is a good time to apply Change Reference to Value since there is only one reference to the new class.

The first thing I need to do is to make the telephone number immutable. I do this by applying *Remove Setting Method (331)* to the fields. The first step of *Remove Setting Method (331)* is to use *Change Function Declaration (124)* to add the two fields to the constructor and enhance the constructor to call the setters.

*class TelephoneNumber...*

```
constructor(areaCode, number) {
  this._areaCode = areaCode;
  this._number = number;
}
```

Now I look at the callers of the setters. For each one, I need to change it to a reassignment. I start with the area code.

*class Person...*

```
get officeAreaCode()    {return this._telephoneNumber.areaCode;}
set officeAreaCode(arg) {
  this._telephoneNumber = new TelephoneNumber(arg, this.officeNumber);
}
get officeNumber()    {return this._telephoneNumber.number;}
set officeNumber(arg) {this._telephoneNumber.number = arg;}
```

I then repeat that step with the remaining field.

*class Person...*

```
get officeAreaCode()    {return this._telephoneNumber.areaCode;}
set officeAreaCode(arg) {
  this._telephoneNumber = new TelephoneNumber(arg, this.officeNumber);
}
get officeNumber()    {return this._telephoneNumber.number;}
set officeNumber(arg) {
  this._telephoneNumber = new TelephoneNumber(this.officeAreaCode, arg);
}
```

Now the telephone number is immutable, it is ready to become a true value. The citizenship test for a value object is that it uses value-based equality. This is an area where JavaScript falls down, as there is nothing in the language and core libraries that understands replacing a reference-based equality with a value-based one. The best I can do is to create my own `equals` method.

*class TelephoneNumber...*

```
equals(other) {
  if (!(other instanceof TelephoneNumber)) return false;
  return this.areaCode === other.areaCode &&
    this.number === other.number;
}
```

It's also important to test it with something like

```
it('telephone equals', function() {
  assert(       new TelephoneNumber("312", "555-0142")
         .equals(new TelephoneNumber("312", "555-0142")));
});
```

*The unusual formatting I use here should make it obvious that they are the same constructor call.*

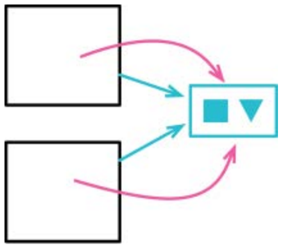
The vital thing I do in the test is create two independent objects and test that they match as equal.

In most object-oriented languages, there is a built-in equality test that is supposed to be overridden for value-based equality. In Ruby, I can override the `==` operator; in Java, I override the `Object.equals()` method. And whenever I override an equality method, I usually need to override a hashcode generating method too (e.g., `Object.hashCode()` in Java) to ensure collections that use hashing work properly with my new value.

If the telephone number is used by more than one client, the procedure is still the same. As I apply *Remove Setting Method (331)*, I'll be modifying several clients instead of just one. Tests for non-equal telephone numbers, as well as comparisons to non-telephone-numbers and null values, are also worthwhile.

## Change Value to Reference

inverse of: *Change Reference to Value (252)*



```
let customer = new Customer(customerData);
```

⇓

```
let customer = customerRepository.get(customerData.id);
```

### Motivation

A data structure may have several records linked to the same logical data structure. I might read in a list of orders, some of which are for the same customer. When I have sharing like this, I can represent it by treating the customer either as a value or as a reference. With a value, the customer data is copied into each order; with a reference, there is only one data structure that multiple orders link to.

If the customer never needs to be updated, then both approaches are reasonable. It is, perhaps, a bit confusing to have multiple copies of the same data, but it's common enough to not be a problem. In some cases, there may be issues with memory due to multiple copies—but, like any performance issue, that's relatively rare.

The biggest difficulty in having physical copies of the same logical data occurs when I need to update the shared data. I then have to find all the copies and update them all. If I miss one, I'll get a troubling inconsistency in my data. In this case, it's often worthwhile to change the copied data into a single reference. That way, any change is visible to all the customer's orders.

Changing a value to a reference results in only one object being present for an entity, and it usually means I need some kind of repository where I can access these objects. I then only create the object for an entity once, and everywhere else I retrieve it from the repository.

## Mechanics

- Create a repository for instances of the related object (if one isn't already present).
- Ensure the constructor has a way of looking up the correct instance of the related object.
- Change the constructors for the host object to use the repository to obtain the related object. Test after each change.

## Example

I'll begin with a class that represents orders, which I might create from an incoming JSON document. Part of the order data is a customer ID from which I'm creating a customer object.

*class Order...*

```
constructor(data) {
  this._number = data.number;
  this._customer = new Customer(data.customer);
  // load other data
}
get customer() {return this._customer;}
```

*class Customer...*

```
constructor(id) {
  this._id = id;
}
get id() {return this._id;}
```

The customer object I create this way is a value. If I have five orders that refer to the customer ID of 123, I'll have five separate customer objects. Any change I make to one of them will not be reflected in the others. Should I want to enrich the customer objects, perhaps by gathering data from a customer service, I'd have to update all five customers with the same data. Having duplicate objects like this always makes me nervous—it's confusing to have multiple objects representing the same entity, such as a customer. This problem is particularly awkward if the customer object is mutable, which can lead to inconsistencies between the customer objects.

If I want to use the same customer object each time, I'll need a place to store it. Exactly where to store entities like this will vary from application to application, but for a simple case I like to use a repository object [mf-repos].

```
let _repositoryData;

export function initialize() {
  _repositoryData = {};
  _repositoryData.customers = new Map();
}

export function registerCustomer(id) {
  if (! _repositoryData.customers.has(id))
    _repositoryData.customers.set(id, new Customer(id));
  return findCustomer(id);
}

export function findCustomer(id) {
  return _repositoryData.customers.get(id);
}
```

The repository allows me to register customer objects with an ID and ensures I only create one customer object with the same ID. With this in place, I can change the order's constructor to use it.

Often, when doing this refactoring, the repository already exists, so I can just use it.

The next step is to figure out how the constructor for the order can obtain the correct customer object. In this case it's easy, since the customer's ID is present in the input data stream.

*class Order…*

```
constructor(data) {
  this._number = data.number;
  this._customer = registerCustomer(data.customer);
  // load other data
}
get customer() {return this._customer;}
```

Now, any changes I make to the customer of one order will be synchronized across all the orders sharing the same customer.

For this example, I created a new customer object with the first order that referenced it. Another common approach is to get a list of customers, populate the repository with them, and then link to them as I read the orders. In that case, an order that contains a customer ID not in the repository would indicate an error.

One problem with this code is that the constructor body is coupled to the global repository. Globals should be treated with care—like a powerful drug, they can be beneficial in small doses but a poison if used too much. If I'm concerned about it, I can pass the repository as a parameter to the constructor.

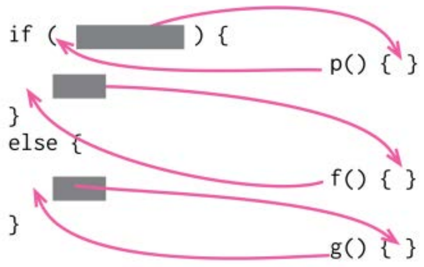
# Chapter 10

# Simplifying Conditional Logic

Much of the power of programs comes from their ability to implement conditional logic—but, sadly, much of the complexity of programs lies in these conditionals. I often use refactoring to make conditional sections easier to understand. I regularly apply *Decompose Conditional (260)* to complicated conditionals, and I use *Consolidate Conditional Expression (263)* to make logical combinations clearer. I use *Replace Nested Conditional with Guard Clauses (266)* to clarify cases where I want to run some pre-checks before my main processing. If I see several conditions using the same switching logic, it's a good time to pull *Replace Conditional with Polymorphism (272)* out the box.

A lot of conditionals are used to handle special cases, such as nulls; if that logic is mostly the same, then *Introduce Special Case (289)* (often referred to as *Introduce Null Object (289)*) can remove a lot of duplicate code. And, although I like to remove conditions a lot, if I want to communicate (and check) a program's state, I find *Introduce Assertion (302)* a worthwhile addition.

## Decompose Conditional



```
if (!aDate.isBefore(plan.summerStart) && !aDate.isAfter(plan.summerEnd))
  charge = quantity * plan.summerRate;
else
  charge = quantity * plan.regularRate + plan.regularServiceCharge;
```

⇓

```
if (summer())
  charge = summerCharge();
else
  charge = regularCharge();
```

### Motivation

One of the most common sources of complexity in a program is complex conditional logic. As I write code to do various things depending on various conditions, I can quickly end up with a pretty long function. Length of a function is in itself a factor that makes it harder to read, but conditions increase the difficulty. The problem usually lies in the fact that the code, both in the condition checks and in the actions, tells me what happens but can easily obscure *why* it happens.

As with any large block of code, I can make my intention clearer by decomposing it and replacing each chunk of code with a function call named after the intention of that chunk. With conditions, I particularly like doing this for the conditional part and each of the alternatives. This way, I highlight the condition and make it clear what I'm branching on. I also highlight the reason for the branching.

This is really just a particular case of applying *Extract Function (106)* to my code, but I like to highlight this case as one where I've often found a remarkably good value for the exercise.

## Mechanics

- Apply *Extract Function (106)* on the condition and each leg of the conditional.

## Example

Suppose I'm calculating the charge for something that has separate rates for winter and summer:

```
if (!aDate.isBefore(plan.summerStart) && !aDate.isAfter(plan.summerEnd))
  charge = quantity * plan.summerRate;
else
  charge = quantity * plan.regularRate + plan.regularServiceCharge;
```

I extract the condition into its own function.

```
if (summer())
  charge = quantity * plan.summerRate;
else
  charge = quantity * plan.regularRate + plan.regularServiceCharge;

function summer() {
  return !aDate.isBefore(plan.summerStart) && !aDate.isAfter(plan.summerEnd);
}
```

Then I do the `then` leg:

```
if (summer())
  charge = summerCharge();
else
  charge = quantity * plan.regularRate + plan.regularServiceCharge;

function summer() {
  return !aDate.isBefore(plan.summerStart) && !aDate.isAfter(plan.summerEnd);
}
function summerCharge() {
  return quantity * plan.summerRate;
}
```

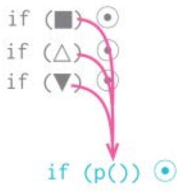
Finally, the `else` leg:

```
if (summer())
  charge = summerCharge();
else
  charge = regularCharge();

function summer() {
  return !aDate.isBefore(plan.summerStart) && !aDate.isAfter(plan.summerEnd);
}
function summerCharge() {
  return quantity * plan.summerRate;
}
function regularCharge() {
  return quantity * plan.regularRate + plan.regularServiceCharge;
}
```

With that done, I like to reformat the conditional using the ternary operator.

```
charge = summer() ? summerCharge() : regularCharge();

function summer() {
  return !aDate.isBefore(plan.summerStart) && !aDate.isAfter(plan.summerEnd);
}
function summerCharge() {
  return quantity * plan.summerRate;
}
function regularCharge() {
  return quantity * plan.regularRate + plan.regularServiceCharge;
}
```

## Consolidate Conditional Expression



```
if (anEmployee.seniority < 2) return 0;
if (anEmployee.monthsDisabled > 12) return 0;
if (anEmployee.isPartTime) return 0;
```

⇓

```
if (isNotEligableForDisability()) return 0;

function isNotEligableForDisability() {
  return ((anEmployee.seniority < 2)
          || (anEmployee.monthsDisabled > 12)
          || (anEmployee.isPartTime));
}
```

### Motivation

Sometimes, I run into a series of conditional checks where each check is different yet the resulting action is the same. When I see this, I use `and` and `or` operators to consolidate them into a single conditional check with a single result.

Consolidating the conditional code is important for two reasons. First, it makes it clearer by showing that I'm really making a single check that combines other checks. The sequence has the same effect, but it looks like I'm carrying out a sequence of separate checks that just happen to be close together. The second reason I like to do this is that it often sets me up for *Extract Function (106)*. Extracting a condition is one of the most useful things I can do to clarify my code. It replaces a statement of what I'm doing with why I'm doing it.

The reasons in favor of consolidating conditionals also point to the reasons against doing it. If I consider it to be truly independent checks that shouldn't be thought of as a single check, I don't do the refactoring.

## Mechanics

- Ensure that none of the conditionals have any side effects.

  If any do, use *Separate Query from Modifier (306)* on them first.

- Take two of the conditional statements and combine their conditions using a logical operator.

  Sequences combine with `or`, nested `if` statements combine with `and`.

- Test.
- Repeat combining conditionals until they are all in a single condition.
- Consider using *Extract Function (106)* on the resulting condition.

## Example

Perusing some code, I see the following:

```
function disabilityAmount(anEmployee) {
  if (anEmployee.seniority < 2) return 0;
  if (anEmployee.monthsDisabled > 12) return 0;
  if (anEmployee.isPartTime) return 0;
  // compute the disability amount
```

It's a sequence of conditional checks which all have the same result. Since the result is the same, I should combine these conditions into a single expression. For a sequence like this, I do it using an `or` operator.

```
function disabilityAmount(anEmployee) {
  if ((anEmployee.seniority < 2)
      || (anEmployee.monthsDisabled > 12)) return 0;
  if (anEmployee.isPartTime) return 0;
  // compute the disability amount
```

I test, then fold in the other condition:

```
function disabilityAmount(anEmployee) {
  if ((anEmployee.seniority < 2)
      || (anEmployee.monthsDisabled > 12)
      || (anEmployee.isPartTime)) return 0;
  // compute the disability amount
```

Once I have them all together, I use *Extract Function (106)* on the condition.

```
function disabilityAmount(anEmployee) {
  if (isNotEligableForDisability()) return 0;
  // compute the disability amount

  function isNotEligableForDisability() {
    return ((anEmployee.seniority < 2)
            || (anEmployee.monthsDisabled > 12)
            || (anEmployee.isPartTime));
  }
```

## Example: Using ands

The example above showed combining statements with an `or`, but I may run into cases that need `ands` as well. Such a case uses nested `if` statements:

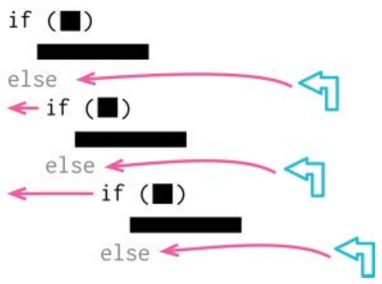
```
if (anEmployee.onVacation)
  if (anEmployee.seniority > 10)
    return 1;
return 0.5;
```

I combine these using `and` operators.

```
if ((anEmployee.onVacation)
    && (anEmployee.seniority > 10)) return 1;
return 0.5;
```

If I have a mix of these, I can combine using `and` and `or` operators as needed. When this happens, things are likely to get messy, so I use *Extract Function (106)* liberally to make it all understandable.

## Replace Nested Conditional with Guard Clauses



```
function getPayAmount() {
  let result;
  if (isDead)
    result = deadAmount();
  else {
    if (isSeparated)
      result = separatedAmount();
    else {
      if (isRetired)
        result = retiredAmount();
      else
        result = normalPayAmount();
    }
  }
  return result;
}
```

```
function getPayAmount() {
  if (isDead) return deadAmount();
  if (isSeparated) return separatedAmount();
  if (isRetired) return retiredAmount();
  return normalPayAmount();
}
```

## Motivation

I often find that conditional expressions come in two styles. In the first style, both legs of the conditional are part of normal behavior, while in the second style, one leg is normal and the other indicates an unusual condition.

These kinds of conditionals have different intentions—and these intentions should come through in the code. If both are part of normal behavior, I use a condition with an `if` and an `else` leg. If the condition is an unusual condition, I check the condition and return if it's true. This kind of check is often called a **guard clause**.

The key point of Replace Nested Conditional with Guard Clauses is emphasis. If I'm using an if-then-else construct, I'm giving equal weight to the `if` leg and the `else` leg. This communicates to the reader that the legs are equally likely and important. Instead, the guard clause says, "This isn't the core to this function, and if it happens, do something and get out."

I often find I use Replace Nested Conditional with Guard Clauses when I'm working with a programmer who has been taught to have only one entry point and one exit point from a method. One entry point is enforced by modern languages, but one exit point is really not a useful rule. Clarity is the key principle: If the method is clearer with one exit point, use one exit point; otherwise don't.

## Mechanics

- Select outermost condition that needs to be replaced, and change it into a guard clause.
- Test.
- Repeat as needed.
- If all the guard clauses return the same result, use *Consolidate Conditional Expression (263)*.

## Example

Here's some code to calculate a payment amount for an employee. It's only relevant if the employee is still with the company, so it has to check for the two other cases.

```
function payAmount(employee) {
  let result;
  if(employee.isSeparated) {
    result = {amount: 0, reasonCode: "SEP"};
  }
  else {
    if (employee.isRetired) {
      result = {amount: 0, reasonCode: "RET"};
    }
    else {
      // logic to compute amount
      lorem.ipsum(dolor.sitAmet);
      consectetur(adipiscing).elit();
      sed.do.eiusmod = tempor.incididunt.ut(labore) && dolore(magna.aliqua);
      ut.enim.ad(minim.veniam);
      result = someFinalComputation();
    }
  }
  return result;
}
```

Nesting the conditionals here masks the true meaning of what it going on. The primary purpose of this code only applies if these conditions aren't the case. In this situation, the intention of the code reads more clearly with guard clauses.

As with any refactoring change, I like to take small steps, so I begin with the topmost condition.

```
function payAmount(employee) {
  let result;
  if (employee.isSeparated) return {amount: 0, reasonCode: "SEP"};
  if (employee.isRetired) {
    result = {amount: 0, reasonCode: "RET"};
  }
  else {
    // logic to compute amount
    lorem.ipsum(dolor.sitAmet);
    consectetur(adipiscing).elit();
    sed.do.eiusmod = tempor.incididunt.ut(labore) && dolore(magna.aliqua);
    ut.enim.ad(minim.veniam);
    result = someFinalComputation();
  }
  return result;
}
```

I test that change and move on to the next one.

```
function payAmount(employee) {
  let result;
  if (employee.isSeparated) return {amount: 0, reasonCode: "SEP"};
  if (employee.isRetired)   return {amount: 0, reasonCode: "RET"};
  // logic to compute amount
  lorem.ipsum(dolor.sitAmet);
  consectetur(adipiscing).elit();
  sed.do.eiusmod = tempor.incididunt.ut(labore) && dolore(magna.aliqua);
  ut.enim.ad(minim.veniam);
  result = someFinalComputation();
  return result;
}
```

At which point the result variable isn't really doing anything useful, so I remove it.

```
function payAmount(employee) {
  ~~let result;~~
  if (employee.isSeparated) return {amount: 0, reasonCode: "SEP"};
  if (employee.isRetired)   return {amount: 0, reasonCode: "RET"};
  // logic to compute amount
  lorem.ipsum(dolor.sitAmet);
  consectetur(adipiscing).elit();
  sed.do.eiusmod = tempor.incididunt.ut(labore) && dolore(magna.aliqua);
  ut.enim.ad(minim.veniam);
  return someFinalComputation();
}
```

The rule is that you always get an extra strawberry when you remove a mutable variable.

## Example: Reversing the Conditions

When reviewing the manuscript of the first edition of this book, Joshua Kerievsky pointed out that we often do Replace Nested Conditional with Guard Clauses by reversing the conditional expressions. Even better, he gave me an example so I didn't have to further tax my imagination.

```
function adjustedCapital(anInstrument) {
  let result = 0;
  if (anInstrument.capital > 0) {
    if (anInstrument.interestRate > 0 && anInstrument.duration > 0) {
      result = (anInstrument.income / anInstrument.duration) * anInstrument.adjustmentFactor;
    }
  }
  return result;
}
```

Again, I make the replacements one at a time, but this time I reverse the condition as I put in the guard clause.

```
function adjustedCapital(anInstrument) {
  let result = 0;
  if (anInstrument.capital <= 0) return result;
  if (anInstrument.interestRate > 0 && anInstrument.duration > 0) {
    result = (anInstrument.income / anInstrument.duration) * anInstrument.adjustmentFactor;
  }
  return result;
}
```

The next conditional is a bit more complicated, so I do it in two steps. First, I simply add a `not`.

```
function adjustedCapital(anInstrument) {
  let result = 0;
  if (anInstrument.capital <= 0) return result;
  if (!(anInstrument.interestRate > 0 && anInstrument.duration > 0)) return result;
  result = (anInstrument.income / anInstrument.duration) * anInstrument.adjustmentFactor;
  return result;
}
```

Leaving `nots` in a conditional like that twists my mind around at a painful angle, so I simplify it:

```
function adjustedCapital(anInstrument) {
  let result = 0;
  if (anInstrument.capital <= 0) return result;
  if (anInstrument.interestRate <= 0 || anInstrument.duration <= 0) return result;
  result = (anInstrument.income / anInstrument.duration) * anInstrument.adjustmentFactor;
  return result;
}
```

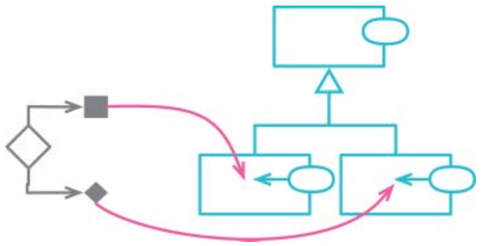
Both of those lines have conditions with the same result, so I apply *Consolidate Conditional Expression (263)*.

```
function adjustedCapital(anInstrument) {
  let result = 0;
  if (   anInstrument.capital      <= 0
      || anInstrument.interestRate <= 0
      || anInstrument.duration     <= 0) return result;
  result = (anInstrument.income / anInstrument.duration) * anInstrument.adjustmentFactor;
  return result;
}
```

The `result` variable is doing two things here. Its first setting to zero indicates what to return when the guard clause triggers; its second value is the final computation. I can get rid of it, which both eliminates its double usage and gets me a strawberry.

```
function adjustedCapital(anInstrument) {
  if (   anInstrument.capital      <= 0
      || anInstrument.interestRate <= 0
      || anInstrument.duration     <= 0) return 0;
  return (anInstrument.income / anInstrument.duration) * anInstrument.adjustmentFactor;
}
```

## Replace Conditional with Polymorphism



```
switch (bird.type) {
  case 'EuropeanSwallow':
    return "average";
  case 'AfricanSwallow':
    return (bird.numberOfCoconuts > 2) ? "tired" : "average";
  case 'NorwegianBlueParrot':
    return (bird.voltage > 100) ? "scorched" : "beautiful";
  default:
    return "unknown";
```

$\Downarrow$

```
class EuropeanSwallow {
  get plumage() {
    return "average";
  }
class AfricanSwallow {
  get plumage() {
     return (this.numberOfCoconuts > 2) ? "tired" : "average";
  }
class NorwegianBlueParrot {
  get plumage() {
     return (this.voltage > 100) ? "scorched" : "beautiful";
  }
```

### Motivation

Complex conditional logic is one of the hardest things to reason about in programming, so I always look for ways to add structure to conditional logic. Often, I find I can separate the logic into different circumstances—high-level cases—to divide the conditions. Sometimes it's enough to represent this division within the structure of a conditional itself, but using classes and polymorphism can make the separation more explicit.

A common case for this is where I can form a set of types, each handling the conditional logic differently. I might notice that books, music, and food vary in how they are handled because of their type. This is made most obvious when there are several functions that have a switch statement on a type code. In that case, I remove the duplication of the common switch logic by creating classes for each case and using polymorphism to bring out the type-specific behavior.

Another situation is where I can think of the logic as a base case with variants. The base case may be the most common or most straightforward. I can put this logic into a superclass which allows me to reason about it without having to worry about the variants. I then put each variant case into a subclass, which I express with code that emphasizes its difference from the base case.

Polymorphism is one of the key features of object-oriented programming—and, like any useful feature, it's prone to overuse. I've come across people who argue that all examples of conditional logic should be replaced with polymorphism. I don't agree with that view. Most of my conditional logic uses basic conditional statements—if/else and switch/case. But when I see complex conditional logic that can be improved as discussed above, I find polymorphism a powerful tool.

## Mechanics

- If classes do not exist for polymorphic behavior, create them together with a factory function to return the correct instance.
- Use the factory function in calling code.
- Move the conditional function to the superclass.

  If the conditional logic is not a self-contained function, use *Extract Function (106)* to make it so.
- Pick one of the subclasses. Create a subclass method that overrides the conditional statement method. Copy the body of that leg of the conditional statement into the subclass method and adjust it to fit.
- Repeat for each leg of the conditional.
- Leave a default case for the superclass method. Or, if superclass should be abstract, declare that method as abstract or throw an error to show it should be the responsibility of a subclass.

## Example

My friend has a collection of birds and wants to know how fast they can fly and what they have for plumage. So we have a couple of small programs to determine the information.

```
function plumages(birds) {
  return new Map(birds.map(b => [b.name, plumage(b)]));
}
function speeds(birds) {
  return new Map(birds.map(b => [b.name, airSpeedVelocity(b)]));
}

function plumage(bird) {
  switch (bird.type) {
  case 'EuropeanSwallow':
    return "average";
  case 'AfricanSwallow':
    return (bird.numberOfCoconuts > 2) ? "tired" : "average";
  case 'NorwegianBlueParrot':
    return (bird.voltage > 100) ? "scorched" : "beautiful";
  default:
    return "unknown";
  }
}

function airSpeedVelocity(bird) {
  switch (bird.type) {
  case 'EuropeanSwallow':
    return 35;
  case 'AfricanSwallow':
    return 40 - 2 * bird.numberOfCoconuts;
  case 'NorwegianBlueParrot':
    return (bird.isNailed) ? 0 : 10 + bird.voltage / 10;
  default:
    return null;
  }
}
```

We have a couple of different operations that vary with the type of bird, so it makes sense to create classes and use polymorphism for any type-specific behavior.

I begin by using *Combine Functions into Class (144)* on `airSpeedVelocity` and `plumage`.

```
function plumage(bird) {
  return new Bird(bird).plumage;
}

function airSpeedVelocity(bird) {
  return new Bird(bird).airSpeedVelocity;
}
```

```
class Bird {
  constructor(birdObject) {
    Object.assign(this, birdObject);
  }
  get plumage() {
    switch (this.type) {
    case 'EuropeanSwallow':
      return "average";
    case 'AfricanSwallow':
      return (this.numberOfCoconuts > 2) ? "tired" : "average";
    case 'NorwegianBlueParrot':
      return (this.voltage > 100) ? "scorched" : "beautiful";
    default:
      return "unknown";
    }
  }
  get airSpeedVelocity() {
    switch (this.type) {
    case 'EuropeanSwallow':
      return 35;
    case 'AfricanSwallow':
      return 40 - 2 * this.numberOfCoconuts;
    case 'NorwegianBlueParrot':
      return (this.isNailed) ? 0 : 10 + this.voltage / 10;
    default:
      return null;
    }
  }
}
```

I now add subclasses for each kind of bird, together with a factory function to instantiate the appropriate subclass.

```
function plumage(bird) {
  return createBird(bird).plumage;
}

function airSpeedVelocity(bird) {
  return createBird(bird).airSpeedVelocity;
}

  function createBird(bird) {
    switch (bird.type) {
    case 'EuropeanSwallow':
      return new EuropeanSwallow(bird);
    case 'AfricanSwallow':
      return new AfricanSwallow(bird);
    case 'NorweigianBlueParrot':
      return new NorwegianBlueParrot(bird);
    default:
      return new Bird(bird);
    }
  }
```

```
class EuropeanSwallow extends Bird {
}

class AfricanSwallow extends Bird {
}

class NorwegianBlueParrot extends Bird {
}
```

Now that I've created the class structure that I need, I can begin on the two conditional methods. I'll begin with plumage. I take one leg of the switch statement and override it in the appropriate subclass.

*class EuropeanSwallow...*

```
get plumage() {
  return "average";
}
```

*class Bird...*

```
get plumage() {
  switch (this.type) {
  case 'EuropeanSwallow':
    throw "oops";
  case 'AfricanSwallow':
    return (this.numberOfCoconuts > 2) ? "tired" : "average";
  case 'NorwegianBlueParrot':
    return (this.voltage > 100) ? "scorched" : "beautiful";
  default:
    return "unknown";
  }
}
```

*I put in the* `throw` *because I'm paranoid.*

I can compile and test at this point. Then, if all is well, I do the next leg.

*class AfricanSwallow...*

```
get plumage() {
   return (this.numberOfCoconuts > 2) ? "tired" : "average";
}
```

Then, the Norwegian Blue:

*class NorwegianBlueParrot...*

```
get plumage() {
   return (this.voltage > 100) ? "scorched" : "beautiful";
}
```

I leave the superclass method for the default case.

*class Bird...*

```
get plumage() {
  return "unknown";
}
```

I repeat the same process for `airSpeedVelocity`. Once I'm done, I end up with the following code (I also inlined the top-level functions for `airSpeedVelocity` and `plumage`):

```
function plumages(birds) {
  return new Map(birds
                 .map(b => createBird(b))
                 .map(bird => [bird.name, bird.plumage]));
}
function speeds(birds) {
  return new Map(birds
                 .map(b => createBird(b))
                 .map(bird => [bird.name, bird.airSpeedVelocity]));
}

function createBird(bird) {
  switch (bird.type) {
  case 'EuropeanSwallow':
    return new EuropeanSwallow(bird);
  case 'AfricanSwallow':
    return new AfricanSwallow(bird);
  case 'NorwegianBlueParrot':
    return new NorwegianBlueParrot(bird);
  default:
    return new Bird(bird);
  }
}

class Bird {
  constructor(birdObject) {
    Object.assign(this, birdObject);
  }
  get plumage() {
    return "unknown";
  }
  get airSpeedVelocity() {
    return null;
  }
}
class EuropeanSwallow extends Bird {
  get plumage() {
    return "average";
  }
  get airSpeedVelocity() {
    return 35;
  }
}
```

```
class AfricanSwallow extends Bird {
  get plumage() {
    return (this.numberOfCoconuts > 2) ? "tired" : "average";
  }
  get airSpeedVelocity() {
    return 40 - 2 * this.numberOfCoconuts;
  }
}
class NorwegianBlueParrot extends Bird {
  get plumage() {
    return (this.voltage > 100) ? "scorched" : "beautiful";
  }
  get airSpeedVelocity() {
    return (this.isNailed) ? 0 : 10 + this.voltage / 10;
  }
}
```

Looking at this final code, I can see that the superclass `Bird` isn't strictly needed. In JavaScript, I don't need a type hierarchy for polymorphism; as long as my objects implement the appropriately named methods, everything works fine. In this situation, however, I like to keep the unnecessary superclass as it helps explain the way the classes are related in the domain.

## Example: Using Polymorphism for Variation

With the birds example, I'm using a clear generalization hierarchy. That's how subclassing and polymorphism is often discussed in textbooks (including mine)—but it's not the only way inheritance is used in practice; indeed, it probably isn't the most common or best way. Another case for inheritance is when I wish to indicate that one object is mostly similar to another, but with some variations.

As an example of this case, consider some code used by a rating agency to compute an investment rating for the voyages of sailing ships. The rating agency gives out either an "A" or "B" rating, depending of various factors due to risk and profit potential. The risk comes from assessing the nature of the voyage as well as the history of the captain's prior voyages.

```
function rating(voyage, history) {
  const vpf = voyageProfitFactor(voyage, history);
  const vr = voyageRisk(voyage);
  const chr = captainHistoryRisk(voyage, history);
  if (vpf * 3 > (vr + chr * 2)) return "A";
  else return "B";
}
```

```
function voyageRisk(voyage) {
  let result = 1;
  if (voyage.length > 4) result += 2;
  if (voyage.length > 8) result += voyage.length - 8;
  if (["china", "east-indies"].includes(voyage.zone)) result += 4;
  return Math.max(result, 0);
}
function captainHistoryRisk(voyage, history) {
  let result = 1;
  if (history.length < 5) result += 4;
  result += history.filter(v => v.profit < 0).length;
  if (voyage.zone === "china" && hasChina(history)) result -= 2;
  return Math.max(result, 0);
}
function hasChina(history) {
  return history.some(v => "china" === v.zone);
}
function voyageProfitFactor(voyage, history) {
  let result = 2;
  if (voyage.zone === "china") result += 1;
  if (voyage.zone === "east-indies") result += 1;
  if (voyage.zone === "china" && hasChina(history)) {
    result += 3;
    if (history.length > 10) result += 1;
    if (voyage.length > 12) result += 1;
    if (voyage.length > 18) result -= 1;
  }
  else {
    if (history.length > 8) result += 1;
    if (voyage.length > 14) result -= 1;
  }
  return result;
}
```

The functions `voyageRisk` and `captainHistoryRisk` score points for risk, `voyageProfitFactor` scores points for the potential profit, and `rating` combines these to give the overall rating for the voyage.

The calling code would look something like this:

```
const voyage = {zone: "west-indies", length: 10};
const history = [
  {zone: "east-indies", profit:  5},
  {zone: "west-indies", profit: 15},
  {zone: "china",       profit: -2},
  {zone: "west-africa", profit:  7},
];

const myRating = rating(voyage, history);
```

What I want to focus on here is how a couple of places use conditional logic to handle the case of a voyage to China where the captain has been to China before.

```
function rating(voyage, history) {
  const vpf = voyageProfitFactor(voyage, history);
  const vr = voyageRisk(voyage);
  const chr = captainHistoryRisk(voyage, history);
  if (vpf * 3 > (vr + chr * 2)) return "A";
  else return "B";
}
  function voyageRisk(voyage) {
  let result = 1;
  if (voyage.length > 4) result += 2;
  if (voyage.length > 8) result += voyage.length - 8;
  if (["china", "east-indies"].includes(voyage.zone)) result += 4;
  return Math.max(result, 0);
}
function captainHistoryRisk(voyage, history) {
  let result = 1;
  if (history.length < 5) result += 4;
  result += history.filter(v => v.profit < 0).length;
  if (voyage.zone === "china" && hasChina(history)) result -= 2;
  return Math.max(result, 0);
}
function hasChina(history) {
  return history.some(v => "china" === v.zone);
}
function voyageProfitFactor(voyage, history) {
  let result = 2;
  if (voyage.zone === "china") result += 1;
  if (voyage.zone === "east-indies") result += 1;
  if (voyage.zone === "china" && hasChina(history)) {
    result += 3;
    if (history.length > 10) result += 1;
    if (voyage.length > 12) result += 1;
    if (voyage.length > 18) result -= 1;
  }
  else {
    if (history.length > 8) result += 1;
    if (voyage.length > 14) result -= 1;
  }
  return result;
}
```

I will use inheritance and polymorphism to separate out the logic for handling these cases from the base logic. This is a particularly useful refactoring if I'm about to introduce more special logic for this case—and the logic for these repeat China voyages can make it harder to understand the base case.

I'm beginning with a set of functions. To introduce polymorphism, I need to create a class structure, so I begin by applying *Combine Functions into Class (144)*. This results in the following code:

```
function rating(voyage, history) {
  return new Rating(voyage, history).value;
}

class Rating {
  constructor(voyage, history) {
    this.voyage = voyage;
    this.history = history;
  }
  get value() {
    const vpf = this.voyageProfitFactor;
    const vr = this.voyageRisk;
    const chr = this.captainHistoryRisk;
    if (vpf * 3 > (vr + chr * 2)) return "A";
    else return "B";
  }
  get voyageRisk() {
    let result = 1;
    if (this.voyage.length > 4) result += 2;
    if (this.voyage.length > 8) result += this.voyage.length - 8;
    if (["china", "east-indies"].includes(this.voyage.zone)) result += 4;
    return Math.max(result, 0);
  }
  get captainHistoryRisk() {
    let result = 1;
    if (this.history.length < 5) result += 4;
    result += this.history.filter(v => v.profit < 0).length;
    if (this.voyage.zone === "china" && this.hasChinaHistory) result -= 2;
    return Math.max(result, 0);
  }
  get voyageProfitFactor() {
    let result = 2;

    if (this.voyage.zone === "china") result += 1;
    if (this.voyage.zone === "east-indies") result += 1;
    if (this.voyage.zone === "china" && this.hasChinaHistory) {
      result += 3;
      if (this.history.length > 10) result += 1;
      if (this.voyage.length > 12) result += 1;
      if (this.voyage.length > 18) result -= 1;
    }
    else {
      if (this.history.length > 8) result += 1;
      if (this.voyage.length > 14) result -= 1;
    }
    return result;
  }
  get hasChinaHistory() {
    return this.history.some(v => "china" === v.zone);
  }
}
```

That's given me the class for the base case. I now need to create an empty subclass to house the variant behavior.

```
class ExperiencedChinaRating extends Rating {
}
```

I then create a factory function to return the variant class when needed.

```
function createRating(voyage, history) {
  if (voyage.zone === "china" && history.some(v => "china" === v.zone))
    return new ExperiencedChinaRating(voyage, history);
  else return new Rating(voyage, history);
}
```

I need to modify any callers to use the factory function instead of directly invoking the constructor, which in this case is just the rating function.

```
function rating(voyage, history) {
  return createRating(voyage, history).value;
}
```

There are two bits of behavior I need to move into a subclass. I begin with the logic in `captainHistoryRisk`:

*class Rating...*

```
get captainHistoryRisk() {
  let result = 1;
  if (this.history.length < 5) result += 4;
  result += this.history.filter(v => v.profit < 0).length;
  if (this.voyage.zone === "china" && this.hasChinaHistory) result -= 2;
  return Math.max(result, 0);
}
```

I write the overriding method in the subclass:

*class ExperiencedChinaRating*

```
get captainHistoryRisk() {
  const result =  super.captainHistoryRisk - 2;
  return Math.max(result, 0);
}
```

*class Rating...*

```
get captainHistoryRisk() {
  let result = 1;
  if (this.history.length < 5) result += 4;
  result += this.history.filter(v => v.profit < 0).length;
  ~~if (this.voyage.zone === "china" && this.hasChinaHistory) result -= 2;~~
  return Math.max(result, 0);
}
```

Separating the variant behavior from `voyageProfitFactor` is a bit more messy. I can't simply remove the variant behavior and call the superclass method since there is an alternative path here. I also don't want to copy the whole superclass method down to the subclass.

*class Rating...*

```
get voyageProfitFactor() {
  let result = 2;

  if (this.voyage.zone === "china") result += 1;
  if (this.voyage.zone === "east-indies") result += 1;
  if (this.voyage.zone === "china" && this.hasChinaHistory) {
    result += 3;
    if (this.history.length > 10) result += 1;
    if (this.voyage.length > 12) result += 1;
    if (this.voyage.length > 18) result -= 1;
  }
  else {
    if (this.history.length > 8) result += 1;
    if (this.voyage.length > 14) result -= 1;
  }
  return result;
}
```

So my response is to first use *Extract Function (106)* on the entire conditional block.

*class Rating...*

```
get voyageProfitFactor() {
  let result = 2;

  if (this.voyage.zone === "china") result += 1;
  if (this.voyage.zone === "east-indies") result += 1;
  result += this.voyageAndHistoryLengthFactor;
  return result;
}
```

```
get voyageAndHistoryLengthFactor() {
  let result = 0;
  if (this.voyage.zone === "china" && this.hasChinaHistory) {
    result += 3;
    if (this.history.length > 10) result += 1;
    if (this.voyage.length > 12) result += 1;
    if (this.voyage.length > 18) result -= 1;
  }
  else {
    if (this.history.length > 8) result += 1;
    if (this.voyage.length > 14) result -= 1;
  }
  return result;
}
```

A function name with an "And" in it is a pretty bad smell, but I'll let it sit and reek for a moment, while I apply the subclassing.

*class Rating...*

```
get voyageAndHistoryLengthFactor() {
  let result = 0;
  if (this.history.length > 8) result += 1;
  if (this.voyage.length > 14) result -= 1;
  return result;
}
```

*class ExperiencedChinaRating...*

```
get voyageAndHistoryLengthFactor() {
  let result = 0;
  result += 3;
  if (this.history.length > 10) result += 1;
  if (this.voyage.length > 12) result += 1;
  if (this.voyage.length > 18) result -= 1;
  return result;
}
```

That's, formally, the end of the refactoring—I've separated the variant behavior out into the subclass. The superclass's logic is simpler to understand and work with, and I only need to deal with variant case when I'm working on the subclass code, which is expressed in terms of its difference with the superclass.

But I feel I should at least outline what I'd do with the awkward new method. Introducing a method purely for overriding by a subclass is a common thing to do when doing this kind of base-and-variation inheritance. But a crude method like this obscures what's going on, instead of revealing.

The "And" gives away that there are really two separate modifications going on here—so I think it's wise to separate them. I'll do this by using *Extract Function*

*(106)* on the history length modification, both in the superclass and subclass. I start with just the superclass:

*class Rating...*

```
get voyageAndHistoryLengthFactor() {
  let result = 0;
  result += this.historyLengthFactor;
  if (this.voyage.length > 14) result -= 1;
  return result;
}
get historyLengthFactor() {
  return (this.history.length > 8) ? 1 : 0;
}
```

I do the same with the subclass:

*class ExperiencedChinaRating...*

```
get voyageAndHistoryLengthFactor() {
  let result = 0;
  result += 3;
  result += this.historyLengthFactor;
  if (this.voyage.length > 12) result += 1;
  if (this.voyage.length > 18) result -= 1;
  return result;
}
get historyLengthFactor() {
  return (this.history.length > 10) ? 1 : 0;
}
```

I can then use *Move Statements to Callers (217)* on the superclass case.

*class Rating...*

```
get voyageProfitFactor() {
  let result = 2;
  if (this.voyage.zone === "china") result += 1;
  if (this.voyage.zone === "east-indies") result += 1;
  result += this.historyLengthFactor;
  result += this.voyageAndHistoryLengthFactor;
  return result;
}

get voyageAndHistoryLengthFactor() {
  let result = 0;
  ~~result += this.historyLengthFactor;~~
  if (this.voyage.length > 14) result -= 1;
  return result;
}
```

*class ExperiencedChinaRating...*

```
get voyageAndHistoryLengthFactor() {
  let result = 0;
  result += 3;
  ~~result += this.historyLengthFactor;~~
  if (this.voyage.length > 12) result += 1;
  if (this.voyage.length > 18) result -= 1;
  return result;
}
```

I'd then use *Rename Function (124)*.

*class Rating...*

```
get voyageProfitFactor() {
  let result = 2;
  if (this.voyage.zone === "china") result += 1;
  if (this.voyage.zone === "east-indies") result += 1;
  result += this.historyLengthFactor;
  result += this.voyageLengthFactor;
  return result;
}

get voyageLengthFactor() {
  return (this.voyage.length > 14) ? - 1: 0;
}
```

*Changing to a ternary to simplify* `voyageLengthFactor`.

*class ExperiencedChinaRating...*

```
get voyageLengthFactor() {
  let result = 0;
  result += 3;
  if (this.voyage.length > 12) result += 1;
  if (this.voyage.length > 18) result -= 1;
  return result;
}
```

One last thing. I don't think adding 3 points makes sense as part of the voyage length factor—it's better added to the overall result.

*class ExperiencedChinaRating...*

```
  get voyageProfitFactor() {
    return super.voyageProfitFactor + 3;
  }

get voyageLengthFactor() {
  let result = 0;
  ~~result += 3;~~
  if (this.voyage.length > 12) result += 1;
  if (this.voyage.length > 18) result -= 1;
  return result;
}
```

At the end of the refactoring, I have the following code. First, there is the basic rating class which can ignore any complications of the experienced China case:

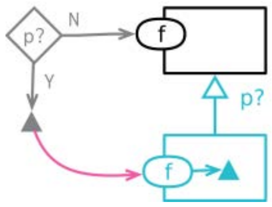
```
class Rating {
  constructor(voyage, history) {
    this.voyage = voyage;
    this.history = history;
  }
  get value() {
    const vpf = this.voyageProfitFactor;
    const vr = this.voyageRisk;
    const chr = this.captainHistoryRisk;
    if (vpf * 3 > (vr + chr * 2)) return "A";
    else return "B";
  }
  get voyageRisk() {
    let result = 1;
    if (this.voyage.length > 4) result += 2;
    if (this.voyage.length > 8) result += this.voyage.length - 8;
    if (["china", "east-indies"].includes(this.voyage.zone)) result += 4;
    return Math.max(result, 0);
  }
  get captainHistoryRisk() {
    let result = 1;
    if (this.history.length < 5) result += 4;
    result += this.history.filter(v => v.profit < 0).length;
    return Math.max(result, 0);
  }
  get voyageProfitFactor() {
    let result = 2;
    if (this.voyage.zone === "china") result += 1;
    if (this.voyage.zone === "east-indies") result += 1;
    result += this.historyLengthFactor;
    result += this.voyageLengthFactor;
    return result;
  }
  get voyageLengthFactor() {
    return (this.voyage.length > 14) ? - 1: 0;
  }
  get historyLengthFactor() {
    return (this.history.length > 8) ? 1 : 0;
  }
}
```

The code for the experienced China case reads as a set of variations on the base:

```
class ExperiencedChinaRating extends Rating {
  get captainHistoryRisk() {
    const result =  super.captainHistoryRisk - 2;
    return Math.max(result, 0);
  }
  get voyageLengthFactor() {
    let result = 0;
    if (this.voyage.length > 12) result += 1;
    if (this.voyage.length > 18) result -= 1;
    return result;
  }
  get historyLengthFactor() {
    return (this.history.length > 10) ? 1 : 0;
  }
  get voyageProfitFactor() {
    return super.voyageProfitFactor + 3;
  }
}
```

## Introduce Special Case

formerly: *Introduce Null Object*



```
if (aCustomer === "unknown") customerName = "occupant";
```

⇓

```
class UnknownCustomer {
    get name() {return "occupant";}
```

### Motivation

A common case of duplicated code is when many users of a data structure check a specific value, and then most of them do the same thing. If I find many parts of the code base having the same reaction to a particular value, I want to bring that reaction into a single place.

A good mechanism for this is the Special Case pattern where I create a special-case element that captures all the common behavior. This allows me to replace most of the special-case checks with simple calls.

A special case can manifest itself in several ways. If all I'm doing with the object is reading data, I can supply a literal object with all the values I need filled in. If I need more behavior than simple values, I can create a special object with methods for all the common behavior. The special-case object can be returned by an encapsulating class, or inserted into a data structure with a transform.

A common value that needs special-case processing is null, which is why this pattern is often called the Null Object pattern. But it's the same approach for any special case—I like to say that Null Object is a special case of Special Case.

### Mechanics

Begin with a container data structure (or class) that contains a property which is the subject of the refactoring. Clients of the container compare the `subject` property of the container to a special-case value. We wish to replace the special-case value of the subject with a special case class or data structure.

- Add a special-case check property to the subject, returning false.
- Create a special-case object with only the special-case check property, returning true.
- Apply *Extract Function (106)* to the special-case comparison code. Ensure that all clients use the new function instead of directly comparing it.
- Introduce the new special-case subject into the code, either by returning it from a function call or by applying a transform function.
- Change the body of the special-case comparison function so that it uses the special-case check property.
- Test.
- Use *Combine Functions into Class (144)* or *Combine Functions into Transform (149)* to move all of the common special-case behavior into the new element.

  Since the special-case class usually returns fixed values to simple requests, these may be handled by making the special case a literal record.

- Use *Inline Function (115)* on the special-case comparison function for the places where it's still needed.

## Example

A utility company installs its services in sites.

*class Site...*

```
  get customer() {return this._customer;}
```

There are various properties of the customer class; I'll consider three of them.

*class Customer...*

```
  get name()            {...}
  get billingPlan()     {...}
  set billingPlan(arg)  {...}
  get paymentHistory()  {...}
```

Most of the time, a site has a customer, but sometimes there isn't one. Someone may have moved out and I don't yet know who, if anyone, has moved in. When this happens, the data record fills the customer field with the string "unknown". Because this can happen, clients of the site need to be able to handle an unknown customer. Here are some example fragments:

*client 1...*

```
const aCustomer = site.customer;
// ... lots of intervening code ...
let customerName;
if (aCustomer === "unknown") customerName = "occupant";
else customerName = aCustomer.name;
```

*client 2...*

```
const plan = (aCustomer === "unknown") ?
      registry.billingPlans.basic
      : aCustomer.billingPlan;
```

*client 3...*

```
if (aCustomer !== "unknown") aCustomer.billingPlan = newPlan;
```

*client 4...*

```
const weeksDelinquent = (aCustomer === "unknown") ?
      0
      : aCustomer.paymentHistory.weeksDelinquentInLastYear;
```

Looking through the code base, I see many clients of the site object that have to deal with an unknown customer. Most of them do the same thing when they get one: They use "occupant" as the name, give them a basic billing plan, and class them as zero-weeks delinquent. This widespread testing for a special case, plus a common response, is what tells me it's time for a Special Case Object.

I begin by adding a method to the customer to indicate it is unknown.

*class Customer...*

```
get isUnknown() {return false;}
```

I then add an Unknown Customer class.

```
class UnknownCustomer {
  get isUnknown() {return true;}
}
```

Note that I don't make `UnknownCustomer` a subclass of `Customer`. In other languages, particularly those statically typed, I would, but JavaScript's rules for subclassing, as well as its dynamic typing, make it better to not do that here.

Now comes the tricky bit. I have to return this new special-case object whenever I expect `"unknown"` and change each test for an unknown value to use the new `isUnknown` method. In general, I always want to arrange things so I can make one small change at a time, then test. But if I change the customer class to return an unknown customer instead of "unknown", I have to make every client testing for "unknown" to call `isUnknown`—and I have to do it all at once. I find that as appealing as eating liver (i.e., not at all).

There is a common technique to use whenever I find myself in this bind. I use *Extract Function (106)* on the code that I'd have to change in lots of places—in this case, the special-case comparison code.

```
function isUnknown(arg) {
  if (!((arg instanceof Customer) || (arg === "unknown")))
    throw new Error(`investigate bad value: <${arg}>`);
  return (arg === "unknown");
}
```

I've put a trap in here for an unexpected value. This can help me to spot any mistakes or odd behavior as I'm doing this refactoring.

I can now use this function whenever I'm testing for an unknown customer. I can change these calls one at a time, testing after each change.

*client 1…*

```
let customerName;
if (isUnknown(aCustomer)) customerName = "occupant";
else customerName = aCustomer.name;
```

After a while, I have done them all.

*client 2…*

```
const plan = (isUnknown(aCustomer)) ?
      registry.billingPlans.basic
      : aCustomer.billingPlan;
```

*client 3…*

```
if (!isUnknown(aCustomer)) aCustomer.billingPlan = newPlan;
```

*client 4…*

```
const weeksDelinquent = isUnknown(aCustomer) ?
      0
      : aCustomer.paymentHistory.weeksDelinquentInLastYear;
```

Once I've changed all the callers to use `isUnknown`, I can change the site class to return an unknown customer.

*class Site…*

```
get customer() {
  return (this._customer === "unknown") ? new UnknownCustomer() :  this._customer;
}
```

I can check that I'm no longer using the "unknown" string by changing `isUnknown` to use the unknown value.

*client 1…*

```
function isUnknown(arg) {
  if (!(arg instanceof Customer || arg instanceof UnknownCustomer))
    throw new Error(`investigate bad value: <${arg}>`);
  return arg.isUnknown;
}
```

I test to ensure that's all working.

Now the fun begins. I can use *Combine Functions into Class (144)* to take each client's special-case check and see if I can replace it with a commonly expected value. At the moment, I have various clients using "occupant" for the name of an unknown customer, like this:

*client 1…*

```
let customerName;
if (isUnknown(aCustomer)) customerName = "occupant";
else customerName = aCustomer.name;
```

I add a suitable method to the unknown customer:

*class UnknownCustomer…*

```
  get name() {return "occupant";}
```

Now I can make all that conditional code go away.

*client 1…*

```
const customerName = aCustomer.name;
```

Once I've tested that this works, I'll probably be able to use *Inline Variable (123)* on that variable too.

Next is the billing plan property.

*client 2…*

```
const plan = (isUnknown(aCustomer)) ?
      registry.billingPlans.basic
      : aCustomer.billingPlan;
```

*client 3…*

```
if (!isUnknown(aCustomer)) aCustomer.billingPlan = newPlan;
```

For read behavior, I do the same thing I did with the name—take the common response and reply with it. With the write behavior, the current code doesn't call the setter for an unknown customer—so for the special case, I let the setter be called, but it does nothing.

*class UnknownCustomer…*

```
  get billingPlan()    {return registry.billingPlans.basic;}
  set billingPlan(arg) { /* ignore */ }
```

*client reader…*

```
const plan = aCustomer.billingPlan;
```

*client writer…*

```
aCustomer.billingPlan = newPlan;
```

Special-case objects are value objects, and thus should always be immutable, even if the objects they are substituting for are not.

The last case is a bit more involved because the special case needs to return another object that has its own properties.

*client…*

```
const weeksDelinquent = isUnknown(aCustomer) ?
      0
      : aCustomer.paymentHistory.weeksDelinquentInLastYear;
```

The general rule with a special-case object is that if it needs to return related objects, they are usually special cases themselves. So here I need to create a null payment history.

*class UnknownCustomer…*

```
  get paymentHistory() {return new NullPaymentHistory();}
```

*class NullPaymentHistory…*

```
  get weeksDelinquentInLastYear() {return 0;}
```

*client…*

```
const weeksDelinquent = aCustomer.paymentHistory.weeksDelinquentInLastYear;
```

I carry on, looking at all the clients to see if I can replace them with the polymorphic behavior. But there will be exceptions—clients that want to do something different with the special case. I may have 23 clients that use "occupant" for the name of an unknown customer, but there's always one that needs something different.

*client…*

```
const name =  ! isUnknown(aCustomer) ? aCustomer.name : "unknown occupant";
```

In that case, I need to retain a special-case check. I will change it to use the method on customer, essentially using *Inline Function (115)* on `isUnknown`.

*client…*

```
const name =  aCustomer.isUnknown ? "unknown occupant" : aCustomer.name;
```

When I'm done with all the clients, I should be able to use *Remove Dead Code (237)* on the global `isPresent` function, as nobody should be calling it any more.

## Example: Using an Object Literal

Creating a class like this is a fair bit of work for what is really a simple value. But for the example I gave, I had to make the class since the customer could be updated. If, however, I only read the data structure, I can use a literal object instead.

Here is the opening case again—just the same, except this time there is no client that updates the customer:

*class Site...*

```
get customer() {return this._customer;}
```

*class Customer...*

```
get name()           {...}
get billingPlan()    {...}
set billingPlan(arg) {...}
get paymentHistory() {...}
```

*client 1...*

```
const aCustomer = site.customer;
// ... lots of intervening code ...
let customerName;
if (aCustomer === "unknown") customerName = "occupant";
else customerName = aCustomer.name;
```

*client 2...*

```
const plan = (aCustomer === "unknown") ?
      registry.billingPlans.basic
      : aCustomer.billingPlan;
```

*client 3...*

```
const weeksDelinquent = (aCustomer === "unknown") ?
      0
      : aCustomer.paymentHistory.weeksDelinquentInLastYear;
```

As with the previous case, I start by adding an `isUnknown` property to the customer and creating a special-case object with that field. The difference is that this time, the special case is a literal.

*class Customer...*

```
get isUnknown() {return false;}
```

*top level…*

```
function createUnknownCustomer() {
  return {
    isUnknown: true,
  };
}
```

I apply *Extract Function (106)* to the special case condition test.

```
function isUnknown(arg) {
  return (arg === "unknown");
}
```

*client 1…*

```
let customerName;
if (isUnknown(aCustomer)) customerName = "occupant";
else customerName = aCustomer.name;
```

*client 2…*

```
const plan = isUnknown(aCustomer) ?
      registry.billingPlans.basic
      : aCustomer.billingPlan;
```

*client 3…*

```
const weeksDelinquent = isUnknown(aCustomer) ?
      0
      : aCustomer.paymentHistory.weeksDelinquentInLastYear;
```

I change the site class and the condition test to work with the special case.

*class Site…*

```
get customer() {
  return (this._customer === "unknown") ?  createUnknownCustomer() :  this._customer;
}
```

*top level…*

```
function isUnknown(arg) {
  return arg.isUnknown;
}
```

Then I replace each standard response with the appropriate literal value. I start with the name:

```
function createUnknownCustomer() {
  return {
    isUnknown: true,
    name: "occupant",
  };
}
```

*client 1...*

```
const customerName = aCustomer.name;
```

Then, the billing plan:

```
function createUnknownCustomer() {
  return {
    isUnknown: true,
    name: "occupant",
    billingPlan: registry.billingPlans.basic,
  };
}
```

*client 2...*

```
const plan = aCustomer.billingPlan;
```

Similarly, I can create a nested null payment history with the literal:

```
function createUnknownCustomer() {
  return {
    isUnknown: true,
    name: "occupant",
    billingPlan: registry.billingPlans.basic,
    paymentHistory: {
      weeksDelinquentInLastYear: 0,
    },
  };
}
```

*client 3...*

```
const weeksDelinquent = aCustomer.paymentHistory.weeksDelinquentInLastYear;
```

If I use a literal like this, I should make it immutable, which I might do with `freeze`. Usually, I'd rather use a class.

## Example: Using a Transform

Both previous cases involve a class, but the same idea can be applied to a record by using a transform step.

Let's assume our input is a simple record structure that looks something like this:

```
{
  name: "Acme Boston",
  location: "Malden MA",
  // more site details
  customer: {
    name: "Acme Industries",
    billingPlan: "plan-451",
    paymentHistory: {
      weeksDelinquentInLastYear: 7
      //more
    },
    // more
  }
}
```

In some cases, the customer isn't known, and such cases are marked in the same way:

```
{
  name: "Warehouse Unit 15",
  location: "Malden MA",
  // more site details
  customer: "unknown",
}
```

I have similar client code that checks for the unknown customer:

*client 1…*

```
const site = acquireSiteData();
const aCustomer = site.customer;
// ... lots of intervening code ...
let customerName;
if (aCustomer === "unknown") customerName = "occupant";
else customerName = aCustomer.name;
```

*client 2…*

```
const plan = (aCustomer === "unknown") ?
      registry.billingPlans.basic
      : aCustomer.billingPlan;
```

*client 3…*

```
const weeksDelinquent = (aCustomer === "unknown") ?
      0
      : aCustomer.paymentHistory.weeksDelinquentInLastYear;
```

My first step is to run the site data structure through a transform that, currently, does nothing but a deep copy.

*client 1…*

```
  const rawSite = acquireSiteData();
  const site = enrichSite(rawSite);
  const aCustomer = site.customer;
  // ... lots of intervening code ...
  let customerName;
  if (aCustomer === "unknown") customerName = "occupant";
  else customerName = aCustomer.name;

function enrichSite(inputSite) {
  return _.cloneDeep(inputSite);
}
```

I apply *Extract Function (106)* to the test for an unknown customer.

```
function isUnknown(aCustomer) {
  return aCustomer === "unknown";
}
```

*client 1…*

```
const rawSite = acquireSiteData();
const site = enrichSite(rawSite);
const aCustomer = site.customer;
// ... lots of intervening code ...
let customerName;
if (isUnknown(aCustomer)) customerName = "occupant";
else customerName = aCustomer.name;
```

*client 2…*

```
const plan = (isUnknown(aCustomer)) ?
      registry.billingPlans.basic
      : aCustomer.billingPlan;
```

*client 3…*

```
const weeksDelinquent = (isUnknown(aCustomer)) ?
      0
      : aCustomer.paymentHistory.weeksDelinquentInLastYear;
```

I begin the enrichment by adding an `isUnknown` property to the customer.

```
function enrichSite(aSite) {
  const result = _.cloneDeep(aSite);
  const unknownCustomer = {
    isUnknown: true,
  };

  if (isUnknown(result.customer)) result.customer = unknownCustomer;
  else result.customer.isUnknown = false;
  return result;
}
```

I can then modify the special-case condition test to include probing for this new property. I keep the original test as well, so that the test will work on both raw and enriched sites.

```
function isUnknown(aCustomer) {
  if (aCustomer === "unknown") return true;
  else return aCustomer.isUnknown;
}
```

I test to ensure that's all OK, then start applying *Combine Functions into Transform (149)* on the special case. First, I move the choice of name into the enrichment function.

```
function enrichSite(aSite) {
  const result = _.cloneDeep(aSite);
  const unknownCustomer = {
    isUnknown: true,
    name: "occupant",
  };

  if (isUnknown(result.customer)) result.customer = unknownCustomer;
  else result.customer.isUnknown = false;
  return result;
}
```

*client 1…*

```
const rawSite = acquireSiteData();
const site = enrichSite(rawSite);
const aCustomer = site.customer;
// ... lots of intervening code ...
const customerName = aCustomer.name;
```

I test, then do the billing plan.

```
function enrichSite(aSite) {
  const result = _.cloneDeep(aSite);
  const unknownCustomer = {
    isUnknown: true,
    name: "occupant",
    billingPlan: registry.billingPlans.basic,
  };

  if (isUnknown(result.customer)) result.customer = unknownCustomer;
  else result.customer.isUnknown = false;
  return result;
}
```

*client 2…*

```
const plan = aCustomer.billingPlan;
```

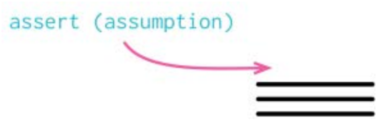
I test again, then do the last client.

```
function enrichSite(aSite) {
  const result = _.cloneDeep(aSite);
  const unknownCustomer = {
    isUnknown: true,
    name: "occupant",
    billingPlan: registry.billingPlans.basic,
    paymentHistory: {
      weeksDelinquentInLastYear: 0,
    }
  };

  if (isUnknown(result.customer)) result.customer = unknownCustomer;
  else result.customer.isUnknown = false;
  return result;
}
```

*client 3…*

```
const weeksDelinquent = aCustomer.paymentHistory.weeksDelinquentInLastYear;
```

## Introduce Assertion



```
if (this.discountRate)
  base = base - (this.discountRate * base);
```

⇓

```
assert(this.discountRate >= 0);
if (this.discountRate)
  base = base - (this.discountRate * base);
```

### Motivation

Often, sections of code work only if certain conditions are true. This may be as simple as a square root calculation only working on a positive input value. With an object, it may require that at least one of a group of fields has a value in it.

Such assumptions are often not stated but can only be deduced by looking through an algorithm. Sometimes, the assumptions are stated with a comment. A better technique is to make the assumption explicit by writing an assertion.

An assertion is a conditional statement that is assumed to be always true. Failure of an assertion indicates a programmer error. Assertion failures should never be checked by other parts of the system. Assertions should be written so that the program functions equally correctly if they are all removed; indeed, some languages provide assertions that can be disabled by a compile-time switch.

I often see people encourage using assertions in order to find errors. While this is certainly a Good Thing, it's not the only reason to use them. I find assertions to be a valuable form of communication—they tell the reader something about the assumed state of the program at this point of execution. I also find them handy for debugging, and their communication value means I'm inclined to leave them in once I've fixed the error I'm chasing. Self-testing code reduces their value for debugging, as steadily narrowing unit tests often do the job better, but I still like assertions for communication.

## Mechanics

- When you see that a condition is assumed to be true, add an assertion to state it.

Since assertions should not affect the running of a system, adding one is always behavior-preserving.

## Example

Here's a simple tale of discounts. A customer can be given a discount rate to apply to all their purchases:

*class Customer…*

```
applyDiscount(aNumber) {
  return (this.discountRate)
    ? aNumber - (this.discountRate * aNumber)
    : aNumber;
}
```

There's an assumption here that the discount rate is a positive number. I can make that assumption explicit by using an assertion. But I can't easily place an assertion into a ternary expression, so first I'll reformulate it as an if-then statement.

*class Customer…*

```
applyDiscount(aNumber) {
  if (!this.discountRate) return aNumber;
  else return aNumber - (this.discountRate * aNumber);
}
```

Now I can easily add the assertion.

*class Customer…*

```
applyDiscount(aNumber) {
  if (!this.discountRate) return aNumber;
  else {
    assert(this.discountRate >= 0);
    return aNumber - (this.discountRate * aNumber);
  }
}
```

In this case, I'd rather put this assertion into the setting method. If the assertion fails in `applyDiscount`, my first puzzle is how it got into the field in the first place.

*class Customer...*

```
set discountRate(aNumber) {
  assert(null === aNumber || aNumber >= 0);
  this._discountRate = aNumber;
}
```

An assertion like this can be particularly valuable if it's hard to spot the error source—which may be an errant minus sign in some input data or some inversion elsewhere in the code.

There is a real danger of overusing assertions. I don't use assertions to check everything that I think is true, but only to check things that *need* to be true. Duplication is a particular problem, as it's common to tweak these kinds of conditions. So I find it's essential to remove any duplication in these conditions, usually by a liberal use of *Extract Function (106)*.

I only use assertions for things that are programmer errors. If I'm reading data from an external source, any value checking should be a first-class part of the program, not an assertion—unless I'm really confident in the external source. Assertions are a last resort to help track bugs—though, ironically, I only use them when I think they should never fail.

Chapter 11

# Refactoring APIs

Modules and their functions are the building blocks of our software. APIs are the joints that we use to plug them together. Making these APIs easy to understand and use is important but also difficult: I need to refactor them as I learn how to improve them.

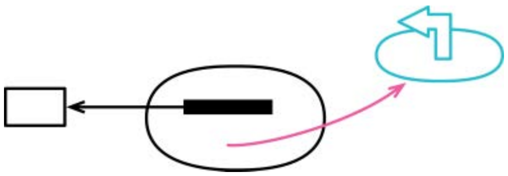
A good API clearly separates any functions that update data from those that only read data. If I see them combined, I use *Separate Query from Modifier (306)* to tease them apart. I can unify functions that only vary due to a value with *Parameterize Function (310)*. Some parameters, however, are really just a signal of an entirely different behavior and are best excised with *Remove Flag Argument (314)*.

Data structures are often unpacked unnecessarily when passed between functions; I prefer to keep them together with *Preserve Whole Object (319)*. Decisions on what should be passed as a parameter, and what can be resolved by the called function, are ones I often need to revisit with *Replace Parameter with Query (324)* and *Replace Query with Parameter (327)*.

A class is a common form of module. I prefer my objects to be as immutable as possible, so I use *Remove Setting Method (331)* whenever I can. Often, when a caller asks for a new object, I need more flexibility than a simple constructor gives, which I can get by using *Replace Constructor with Factory Function (334)*.

The last two refactorings address the difficulty of breaking down a particularly complex function that passes a lot of data around. I can turn that function into an object with *Replace Function with Command (337)*, which makes it easier to use *Extract Function (106)* on the function's body. If I later simplify the function and no longer need it as a command object, I turn it back into a function with *Replace Command with Function (344)*.

## Separate Query from Modifier



```
function getTotalOutstandingAndSendBill() {
  const result = customer.invoices.reduce((total, each) => each.amount + total, 0);
  sendBill();
  return result;
}
```

⇓

```
function totalOutstanding() {
  return customer.invoices.reduce((total, each) => each.amount + total, 0);
}
function sendBill() {
  emailGateway.send(formatBill(customer));
}
```

### Motivation

When I have a function that gives me a value and has no observable side effects, I have a very valuable thing. I can call this function as often as I like. I can move the call to other places in a calling function. It's easier to test. In short, I have a lot less to worry about.

It is a good idea to clearly signal the difference between functions with side effects and those without. A good rule to follow is that any function that returns a value should not have observable side effects—the command-query separation [mf-cqs]. Some programmers treat this as an absolute rule. I'm not 100 percent pure on this (as on anything), but I try to follow it most of the time, and it has served me well.

If I come across a method that returns a value but also has side effects, I always try to separate the query from the modifier.

Note that I use the phrase *observable* side effects. A common optimization is to cache the value of a query in a field so that repeated calls go quicker. Although this changes the state of the object with the cache, the change is not observable. Any sequence of queries will always return the same results for each query.

## Mechanics

- Copy the function, name it as a query.

  Look into the function to see what is returned. If the query is used to populate a variable, the variable's name should provide a good clue.

- Remove any side effects from the new query function.
- Run static checks.
- Find each call of the original method. If that call uses the return value, replace the original call with a call to the query and insert a call to the original method below it. Test after each change.
- Remove return values from original.
- Test.

Often after doing this there will be duplication between the query and the original method that can be tidied up.

## Example

Here is a function that scans a list of names for a miscreant. If it finds one, it returns the name of the bad guy and sets off the alarms. It only does this for the first miscreant it finds (I guess one is enough).

```
function alertForMiscreant (people) {
  for (const p of people) {
    if (p === "Don") {
      setOffAlarms();
      return "Don";
    }
    if (p === "John") {
      setOffAlarms();
      return "John";
    }
  }
  return "";
}
```

I begin by copying the function, naming it after the query aspect of the function.

```
function findMiscreant (people) {
  for (const p of people) {
    if (p === "Don") {
      setOffAlarms();
      return "Don";
    }
    if (p === "John") {
      setOffAlarms();
      return "John";
    }
  }
  return "";
}
```

I remove the side effects from this new query.

```
function findMiscreant (people) {
  for (const p of people) {
    if (p === "Don") {
      ~~setOffAlarms();~~
      return "Don";
    }
    if (p === "John") {
      ~~setOffAlarms();~~
      return "John";
    }
  }
  return "";
}
```

I now go to each caller and replace it with a call to the query, followed by a call to the modifier. So

```
const found = alertForMiscreant(people);
```

changes to

```
const found = findMiscreant(people);
alertForMiscreant(people);
```

I now remove the return values from the modifier.

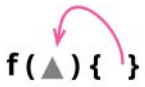
```
function alertForMiscreant (people) {
  for (const p of people) {
    if (p === "Don") {
      setOffAlarms();
      return;
    }
    if (p === "John") {
      setOffAlarms();
      return;
    }
  }
  return;
}
```

Now I have a lot of duplication between the original modifier and the new query, so I can use *Substitute Algorithm (195)* so that the modifier uses the query.

```
function alertForMiscreant (people) {
  if (findMiscreant(people) !== "") setOffAlarms();
}
```

## Parameterize Function

formerly: *Parameterize Method*



```
function tenPercentRaise(aPerson) {
  aPerson.salary = aPerson.salary.multiply(1.1);
}
function fivePercentRaise(aPerson) {
  aPerson.salary = aPerson.salary.multiply(1.05);
}
```

```
function raise(aPerson, factor) {
  aPerson.salary = aPerson.salary.multiply(1 + factor);
}
```

### Motivation

If I see two functions that carry out very similar logic with different literal values, I can remove the duplication by using a single function with parameters for the different values. This increases the usefulness of the function, since I can apply it elsewhere with different values.

### Mechanics

- Select one of the similar methods.
- Use *Change Function Declaration (124)* to add any literals that need to turn into parameters.
- For each caller of the function, add the literal value.
- Test.
- Change the body of the function to use the new parameters. Test after each change.
- For each similar function, replace the call with a call to the parameterized function. Test after each one.

If the original parameterized function doesn't work for a similar function, adjust it for the new function before moving on to the next.

## Example

An obvious example is something like this:

```
function tenPercentRaise(aPerson) {
  aPerson.salary = aPerson.salary.multiply(1.1);
}
function fivePercentRaise(aPerson) {
  aPerson.salary = aPerson.salary.multiply(1.05);
}
```

Hopefully it's obvious that I can replace these with

```
function raise(aPerson, factor) {
  aPerson.salary = aPerson.salary.multiply(1 + factor);
}
```

But it can be a bit more involved than that. Consider this code:

```
function baseCharge(usage) {
  if (usage < 0) return usd(0);
  const amount =
        bottomBand(usage) * 0.03
        + middleBand(usage) * 0.05
        + topBand(usage) * 0.07;
  return usd(amount);
}

function bottomBand(usage) {
  return Math.min(usage, 100);
}

function middleBand(usage) {
  return usage > 100 ? Math.min(usage, 200) - 100 : 0;
}

function topBand(usage) {
  return usage > 200 ? usage - 200 : 0;
}
```

Here the logic is clearly pretty similar—but is it similar enough to support creating a parameterized method for the bands? It is, but may be a touch less obvious than the trivial case above.

When looking to parameterize some related functions, my approach is to take one of the functions and add parameters to it, with an eye to the other cases. With range-oriented things like this, usually the place to start is with the middle range. So I'll work on `middleBand` to change it to use parameters, and then adjust other callers to fit.

middleBand uses two literal values: 100 and 200. These represent the bottom and top of this middle band. I begin by using *Change Function Declaration (124)* to add them to the call. While I'm at it, I'll also change the name of the function to something that makes sense with the parameterization.

```
function withinBand(usage, bottom, top) {
  return usage > 100 ? Math.min(usage, 200) - 100 : 0;
}

function baseCharge(usage) {
  if (usage < 0) return usd(0);
  const amount =
        bottomBand(usage) * 0.03
        + withinBand(usage, 100, 200) * 0.05
        + topBand(usage) * 0.07;
  return usd(amount);
}
```

I replace each literal with a reference to the parameter:

```
function withinBand(usage, bottom, top) {
  return usage > bottom ? Math.min(usage, 200) - bottom : 0;
}
```

then:

```
function withinBand(usage, bottom, top) {
  return usage > bottom ? Math.min(usage, top) - bottom : 0;
}
```

I replace the call to the bottom band with a call to the newly parameterized function.

```
function baseCharge(usage) {
  if (usage < 0) return usd(0);
  const amount =
        withinBand(usage, 0, 100) * 0.03
        + withinBand(usage, 100, 200) * 0.05
        + topBand(usage) * 0.07;
  return usd(amount);
}
```

~~function bottomBand(usage) {~~
~~  return Math.min(usage, 100);~~
~~}~~

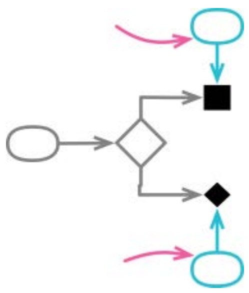
To replace the call to the top band, I need to make use of infinity.

```
function baseCharge(usage) {
  if (usage < 0) return usd(0);
  const amount =
        withinBand(usage, 0, 100) * 0.03
        + withinBand(usage, 100, 200) * 0.05
        + withinBand(usage, 200, Infinity) * 0.07;
  return usd(amount);
}

~~function topBand(usage) {~~
  ~~return usage > 200 ? usage - 200 : 0;~~
~~}~~
```

With the logic working the way it does now, I could remove the initial guard clause. But although it's logically unnecessary now, I like to keep it as it documents how to handle that case.

## Remove Flag Argument

formerly: *Replace Parameter with Explicit Methods*



```
function setDimension(name, value) {
  if (name === "height") {
    this._height = value;
    return;
  }
  if (name === "width") {
    this._width = value;
    return;
  }
}
```

⇓

```
function setHeight(value) {this._height = value;}
function setWidth (value) {this._width = value;}
```

### Motivation

A flag argument is a function argument that the caller uses to indicate which logic the called function should execute. I may call a function that looks like this:

```
function bookConcert(aCustomer, isPremium) {
  if (isPremium) {
    // logic for premium booking
  } else {
    // logic for regular booking
  }
}
```

To book a premium concert, I issue the call like so:

```
bookConcert(aCustomer, true);
```

Flag arguments can also come as enums:

```
bookConcert(aCustomer, CustomerType.PREMIUM);
```

or strings (or symbols in languages that use them):

```
bookConcert(aCustomer, "premium");
```

I dislike flag arguments because they complicate the process of understanding what function calls are available and how to call them. My first route into an API is usually the list of available functions, and flag arguments hide the differences in the function calls that are available. Once I select a function, I have to figure out what values are available for the flag arguments. Boolean flags are even worse since they don't convey their meaning to the reader—in a function call, I can't figure out what `true` means. It's clearer to provide an explicit function for the task I want to do.

```
premiumBookConcert(aCustomer);
```

Not all arguments like this are flag arguments. To be a flag argument, the callers must be setting the boolean value to a literal value, not data that's flowing through the program. Also, the implementation function must be using the argument to influence its control flow, not as data that it passes to further functions.

Removing flag arguments doesn't just make the code clearer—it also helps my tooling. Code analysis tools can now more easily see the difference between calling the premium logic and calling regular logic.

Flag arguments can have a place if there's more than one of them in the function, since otherwise I would need explicit functions for every combination of their values. But that's also a signal of a function doing too much, and I should look for a way to create simpler functions that I can compose for this logic.

## Mechanics

- Create an explicit function for each value of the parameter.

  If the main function has a clear dispatch conditional, use *Decompose Conditional (260)* to create the explicit functions. Otherwise, create wrapping functions.

- For each caller that uses a literal value for the parameter, replace it with a call to the explicit function.

## Example

Looking through some code, I see calls to calculate a delivery date for a shipment. Some of the calls look like

```
aShipment.deliveryDate = deliveryDate(anOrder, true);
```

and some look like

```
aShipment.deliveryDate = deliveryDate(anOrder, false);
```

Faced with code like this, I immediately begin to wonder about the meaning of the boolean value. What is it doing?

The body of `deliveryDate` looks like this:

```
function deliveryDate(anOrder, isRush) {
  if (isRush) {
    let deliveryTime;
    if (["MA", "CT"]     .includes(anOrder.deliveryState)) deliveryTime = 1;
    else if (["NY", "NH"].includes(anOrder.deliveryState)) deliveryTime = 2;
    else deliveryTime = 3;
    return anOrder.placedOn.plusDays(1 + deliveryTime);
  }
  else {
    let deliveryTime;
    if (["MA", "CT", "NY"].includes(anOrder.deliveryState)) deliveryTime = 2;
    else if (["ME", "NH"] .includes(anOrder.deliveryState)) deliveryTime = 3;
    else deliveryTime = 4;
    return anOrder.placedOn.plusDays(2 + deliveryTime);
  }
}
```

Here, the caller is using a literal boolean value to determine which code should run—a classic flag argument. But the whole point of using a function is to follow the caller's instructions, so it is better to clarify the caller's intent with explicit functions.

In this case, I can do this by using *Decompose Conditional (260)*, which gives me this:

```
function deliveryDate(anOrder, isRush) {
  if (isRush) return rushDeliveryDate(anOrder);
  else        return regularDeliveryDate(anOrder);
}
```

```
function rushDeliveryDate(anOrder) {
    let deliveryTime;
    if (["MA", "CT"]     .includes(anOrder.deliveryState)) deliveryTime = 1;
    else if (["NY", "NH"].includes(anOrder.deliveryState)) deliveryTime = 2;
    else deliveryTime = 3;
    return anOrder.placedOn.plusDays(1 + deliveryTime);
}
function regularDeliveryDate(anOrder) {
    let deliveryTime;
    if (["MA", "CT", "NY"].includes(anOrder.deliveryState)) deliveryTime = 2;
    else if (["ME", "NH"] .includes(anOrder.deliveryState)) deliveryTime = 3;
    else deliveryTime = 4;
    return anOrder.placedOn.plusDays(2 + deliveryTime);
}
```

The two new functions capture the intent of the call better, so I can replace each call of

```
aShipment.deliveryDate = deliveryDate(anOrder, true);
```

with

```
aShipment.deliveryDate = rushDeliveryDate(anOrder);
```

and similarly with the other case.

When I've replaced all the callers, I remove `deliveryDate`.

A flag argument isn't just the presence of a boolean value; it's that the boolean is set with a literal rather than data. If all the callers of `deliveryDate` were like this:

```
const isRush = determineIfRush(anOrder);
aShipment.deliveryDate = deliveryDate(anOrder, isRush);
```

then I'd have no problem with `deliveryDate`'s signature (although I'd still want to apply *Decompose Conditional (260)*).

It may be that some callers use the argument as a flag argument by setting it with a literal, while others set the argument with data. In this case, I'd still use Remove Flag Argument, but not change the data callers and not remove `deliveryDate` at the end. That way I support both interfaces for the different uses.

Decomposing the conditional like this is a good way to carry out this refactoring, but it only works if the dispatch on the parameter is the outer part of the function (or I can easily refactor it to make it so). It's also possible that the parameter is used in a much more tangled way, such as this alternative version of `deliveryDate`:

```
function deliveryDate(anOrder, isRush) {
  let result;
  let deliveryTime;
  if (anOrder.deliveryState === "MA" || anOrder.deliveryState === "CT")
    deliveryTime = isRush? 1 : 2;
  else if (anOrder.deliveryState === "NY" || anOrder.deliveryState === "NH") {
    deliveryTime = 2;
    if (anOrder.deliveryState === "NH" && !isRush)
      deliveryTime = 3;
  }
  else if (isRush)
    deliveryTime = 3;
  else if (anOrder.deliveryState === "ME")
    deliveryTime = 3;
  else
    deliveryTime = 4;
  result = anOrder.placedOn.plusDays(2 + deliveryTime);
  if (isRush) result = result.minusDays(1);
  return result;
}
```

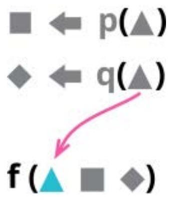
In this case, teasing out `isRush` into a top-level dispatch conditional is likely more work than I fancy. So instead, I can layer functions over the `deliveryDate`:

```
function rushDeliveryDate   (anOrder) {return deliveryDate(anOrder, true);}
function regularDeliveryDate(anOrder) {return deliveryDate(anOrder, false);}
```

These wrapping functions are essentially partial applications of `deliveryDate`, although they are defined in program text rather than by composition of functions.

I can then do the same replacement of callers that I did with the decomposed conditional earlier on. If there aren't any callers using the parameter as data, I like to restrict its visibility or rename it to a name that conveys that it shouldn't be used directly (e.g., `deliveryDateHelperOnly`).

## Preserve Whole Object



```
const low = aRoom.daysTempRange.low;
const high = aRoom.daysTempRange.high;
if (aPlan.withinRange(low, high))
```

⇓

```
if (aPlan.withinRange(aRoom.daysTempRange))
```

### Motivation

If I see code that derives a couple of values from a record and then passes these values into a function, I like to replace those values with the whole record itself, letting the function body derive the values it needs.

Passing the whole record handles change better should the called function need more data from the whole in the future—that change would not require me to alter the parameter list. It also reduces the size of the parameter list, which usually makes the function call easier to understand. If many functions are called with the parts, they often duplicate the logic that manipulates these parts—logic that can often be moved to the whole.

The main reason I wouldn't do this is if I don't want the called function to have a dependency on the whole—which typically occurs when they are in different modules.

Pulling several values from an object to do some logic on them alone is a smell (*Feature Envy (77)*), and usually a signal that this logic should be moved into the whole itself. Preserve Whole Object is particularly common after I've done *Introduce Parameter Object (140)*, as I hunt down any occurrences of the original data clump to replace them with the new object.

If several bits of code only use the same subset of an object's features, then that may indicate a good opportunity for *Extract Class (182)*.

One case that many people miss is when an object calls another object with several of its own data values. If I see this, I can replace those values with a self-reference (`this` in JavaScript).

## Mechanics

- Create an empty function with the desired parameters.

  Give the function an easily searchable name so it can be replaced at the end.

- Fill the body of the new function with a call to the old function, mapping from the new parameters to the old ones.
- Run static checks.
- Adjust each caller to use the new function, testing after each change.

  This may mean that some code that derives the parameter isn't needed, so can fall to *Remove Dead Code (237)*.

- Once all original callers have been changed, use *Inline Function (115)* on the original function.
- Change the name of the new function and all its callers.

## Example

Consider a room monitoring system. It compares its daily temperature range with a range in a predefined heating plan.

*caller…*

```
const low = aRoom.daysTempRange.low;
const high = aRoom.daysTempRange.high;
if (!aPlan.withinRange(low, high))
  alerts.push("room temperature went outside range");
```

*class HeatingPlan…*

```
withinRange(bottom, top) {
  return (bottom >= this._temperatureRange.low) && (top <= this._temperatureRange.high);
}
```

Instead of unpacking the range information when I pass it in, I can pass in the whole range object.

I begin by stating the interface I want as an empty function.

*class HeatingPlan…*

```
xxNEWwithinRange(aNumberRange) {
}
```

Since I intend it to replace the existing `withinRange`, I name it the same but with an easily replaceable prefix.

I then add the body of the function, which relies on calling the existing `withinRange`. The body thus consists of a mapping from the new parameter to the existing ones.

*class HeatingPlan...*

```
xxNEWwithinRange(aNumberRange) {
  return this.withinRange(aNumberRange.low, aNumberRange.high);
}
```

Now I can begin the serious work, taking the existing function calls and having them call the new function.

*caller...*

```
const low = aRoom.daysTempRange.low;
const high = aRoom.daysTempRange.high;
if (!aPlan.xxNEWwithinRange(aRoom.daysTempRange))
  alerts.push("room temperature went outside range");
```

When I've changed the calls, I may see that some of the earlier code isn't needed anymore, so I wield *Remove Dead Code (237)*.

*caller...*

```
~~const low = aRoom.daysTempRange.low;~~
~~const high = aRoom.daysTempRange.high;~~
if (!aPlan.xxNEWwithinRange(aRoom.daysTempRange))
  alerts.push("room temperature went outside range");
```

I replace these one at a time, testing after each change.

Once I've replaced them all, I can use *Inline Function (115)* on the original function.

*class HeatingPlan...*

```
xxNEWwithinRange(aNumberRange) {
  return (aNumberRange.low >= this._temperatureRange.low) &&
    (aNumberRange.high <= this._temperatureRange.high);
}
```

And I finally remove that ugly prefix from the new function and all its callers. The prefix makes it a simple global replace, even if I don't have a robust rename support in my editor.

*class HeatingPlan...*

```
withinRange(aNumberRange) {
  return (aNumberRange.low >= this._temperatureRange.low) &&
    (aNumberRange.high <= this._temperatureRange.high);
}
```

*caller...*

```
if (!aPlan.withinRange(aRoom.daysTempRange))
  alerts.push("room temperature went outside range");
```

## Example: A Variation to Create the New Function

In the above example, I wrote the code for the new function directly. Most of the time, that's pretty simple and the easiest way to go. But there is a variation on this that's occasionally useful—which can allow me to compose the new function entirely from refactorings.

I start with a caller of the existing function.

*caller…*

```
const low = aRoom.daysTempRange.low;
const high = aRoom.daysTempRange.high;
if (!aPlan.withinRange(low, high))
  alerts.push("room temperature went outside range");
```

I want to rearrange the code so I can create the new function by using *Extract Function (106)* on some existing code. The caller code isn't quite there yet, but I can get there by using *Extract Variable (119)* a few times. First, I disentangle the call to the old function from the conditional.

*caller…*

```
const low = aRoom.daysTempRange.low;
const high = aRoom.daysTempRange.high;
const isWithinRange = aPlan.withinRange(low, high);
if (!isWithinRange)
  alerts.push("room temperature went outside range");
```

I then extract the input parameter.

*caller…*

```
const tempRange = aRoom.daysTempRange;
const low = tempRange.low;
const high = tempRange.high;
const isWithinRange = aPlan.withinRange(low, high);
if (!isWithinRange)
  alerts.push("room temperature went outside range");
```

With that done, I can now use *Extract Function (106)* to create the new function.

*caller…*

```
const tempRange = aRoom.daysTempRange;
const isWithinRange = xxNEWwithinRange(aPlan, tempRange);
if (!isWithinRange)
  alerts.push("room temperature went outside range");
```

*top level…*

```
function xxNEWwithinRange(aPlan, tempRange) {
  const low = tempRange.low;
  const high = tempRange.high;
  const isWithinRange = aPlan.withinRange(low, high);
  return isWithinRange;
}
```

Since the original function is in a different context (the `HeatingPlan` class), I need to use *Move Function (198)*.

*caller…*

```
const tempRange = aRoom.daysTempRange;
const isWithinRange = aPlan.xxNEWwithinRange(tempRange);
if (!isWithinRange)
  alerts.push("room temperature went outside range");
```

*class HeatingPlan…*

```
  xxNEWwithinRange(tempRange) {
   const low = tempRange.low;
   const high = tempRange.high;
   const isWithinRange = this.withinRange(low, high);
   return isWithinRange;
  }
```

I then continue as before, replacing other callers and inlining the old function into the new one. I would also inline the variables I extracted to provide the clean separation for extracting the new function.

Because this variation is entirely composed of refactorings, it's particularly handy when I have a refactoring tool with robust extract and inline operations.

## Replace Parameter with Query

formerly: *Replace Parameter with Method*
inverse of: *Replace Query with Parameter (327)*

```
availableVacation(anEmployee, anEmployee.grade);

function availableVacation(anEmployee, grade) {
  // calculate vacation...
```

⇓

```
availableVacation(anEmployee)

function availableVacation(anEmployee) {
  const grade = anEmployee.grade;
  // calculate vacation...
```

### Motivation

The parameter list to a function should summarize the points of variability of that function, indicating the primary ways in which that function may behave differently. As with any statement in code, it's good to avoid any duplication, and it's easier to understand if the parameter list is short.

If a call passes in a value that the function can just as easily determine for itself, that's a form of duplication—one that unnecessarily complicates the caller which has to determine the value of a parameter when it could be freed from that work.

The limit on this is suggested by the phrase "just as easily." By removing the parameter, I'm shifting the responsibility for determining the parameter value. When the parameter is present, determining its value is the caller's responsibility; otherwise, that responsibility shifts to the function body. My usual habit is to simplify life for callers, which implies moving responsibility to the function body—but only if that responsibility is appropriate there.

The most common reason to avoid Replace Parameter with Query is if removing the parameter adds an unwanted dependency to the function body—forcing it to access a program element that I'd rather it remained ignorant of. This may be a new dependency, or an existing one that I'd like to remove. Usually this comes up where I'd need to add a problematic function call to the function body, or access something within a receiver object that I'd prefer to move out later.

The safest case for Replace Parameter with Query is when the value of the parameter I want to remove is determined merely by querying another parameter in the list. There's rarely any point in passing two parameters if one can be determined from the other.

One thing to watch out for is if the function I'm looking at has referential transparency—that is, if I can be sure that it will behave the same way whenever it's called with the same parameter values. Such functions are much easier to reason about and test, and I don't want to alter them to lose that property. So I wouldn't replace a parameter with an access to a mutable global variable.

## Mechanics

- If necessary, use *Extract Function (106)* on the calculation of the parameter.
- Replace references to the parameter in the function body with references to the expression that yields the parameter. Test after each change.
- Use *Change Function Declaration (124)* to remove the parameter.

## Example

I most often use Replace Parameter with Query when I've done some other refactorings that make a parameter no longer needed. Consider this code.

*class Order…*

```
get finalPrice() {
  const basePrice = this.quantity * this.itemPrice;
  let discountLevel;
  if (this.quantity > 100) discountLevel = 2;
  else discountLevel = 1;
  return this.discountedPrice(basePrice, discountLevel);
}

discountedPrice(basePrice, discountLevel) {
  switch (discountLevel) {
    case 1: return basePrice * 0.95;
    case 2: return basePrice * 0.9;
  }
}
```

When I'm simplifying a function, I'm keen to apply *Replace Temp with Query (178)*, which would lead me to *class Order…*

```
get finalPrice() {
  const basePrice = this.quantity * this.itemPrice;
  return this.discountedPrice(basePrice, this.discountLevel);
}

get discountLevel() {
  return (this.quantity > 100) ? 2 : 1;
}
```

Once I've done this, there's no need to pass the result of `discountLevel` to `discountedPrice`—it can just as easily make the call itself.

I replace any reference to the parameter with a call to the method instead.

*class Order…*

```
discountedPrice(basePrice, discountLevel) {
  switch (this.discountLevel) {
    case 1: return basePrice * 0.95;
    case 2: return basePrice * 0.9;
  }
}
```

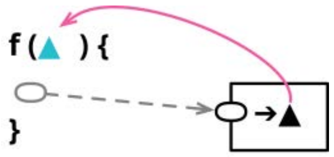
I can then use *Change Function Declaration (124)* to remove the parameter.

*class Order…*

```
get finalPrice() {
  const basePrice = this.quantity * this.itemPrice;
  return this.discountedPrice(basePrice~~, this.discountLevel~~);
}

discountedPrice(basePrice~~, discountLevel~~) {
  switch (this.discountLevel) {
    case 1: return basePrice * 0.95;
    case 2: return basePrice * 0.9;
  }
}
```

## Replace Query with Parameter

inverse of: *Replace Parameter with Query (324)*



```
targetTemperature(aPlan)

function targetTemperature(aPlan) {
  currentTemperature = thermostat.currentTemperature;
  // rest of function...
```

$\Downarrow$

```
targetTemperature(aPlan, thermostat.currentTemperature)

function targetTemperature(aPlan, currentTemperature) {
  // rest of function...
```

### Motivation

When looking through a function's body, I sometimes see references to something in the function's scope that I'm not happy with. This might be a reference to a global variable, or to an element in the same module that I intend to move away. To resolve this, I need to replace the internal reference with a parameter, shifting the responsibility of resolving the reference to the caller of the function.

Most of these cases are due to my wish to alter the dependency relationships in the code—to make the target function no longer dependent on the element I want to parameterize. There's a tension here between converting everything to parameters, which results in long repetitive parameter lists, and sharing a lot of scope which can lead to a lot of coupling between functions. Like most tricky decisions, it's not something I can reliably get right, so it's important that I can reliably change things so the program can take advantage of my increasing understanding.

It's easier to reason about a function that will always give the same result when called with same parameter values—this is called referential transparency. If a function accesses some element in its scope that isn't referentially transparent, then the containing function also lacks referential transparency. I can fix that by moving that element to a parameter. Although such a move will shift responsibility to the caller, there is often a lot to be gained by creating clear modules with referential transparency. A common pattern is to have modules consisting of pure functions which are wrapped by logic that handles the I/O and other variable elements of a program. I can use Replace Query with Parameter to purify parts of a program, making those parts easier to test and reason about.

But Replace Query with Parameter isn't just a bag of benefits. By moving a query to a parameter, I force my caller to figure out how to provide this value. This complicates life for callers of the functions, and my usual bias is to design interfaces that make life easier for their consumers. In the end, it boils down to allocation of responsibility around the program, and that's a decision that's neither easy nor immutable—which is why this refactoring (and its inverse) is one that I need to be very familiar with.

## Mechanics

- Use *Extract Variable (119)* on the query code to separate it from the rest of the function body.
- Apply *Extract Function (106)* to the body code that isn't the call to the query.

  Give the new function an easily searchable name, for later renaming.
- Use *Inline Variable (123)* to get rid of the variable you just created.
- Apply *Inline Function (115)* to the original function.
- Rename the new function to that of the original.

## Example

Consider a simple, yet annoying, control system for temperature. It allows the user to select a temperature on a thermostat—but only sets the target temperature within a range determined by a heating plan.

*class HeatingPlan…*

```
get targetTemperature() {
  if      (thermostat.selectedTemperature >  this._max) return this._max;
  else if (thermostat.selectedTemperature <  this._min) return this._min;
  else return thermostat.selectedTemperature;
}
```

*caller…*

```
if      (thePlan.targetTemperature > thermostat.currentTemperature) setToHeat();
else if (thePlan.targetTemperature < thermostat.currentTemperature) setToCool();
else setOff();
```

As a user of such a system, I might be annoyed to have my desires overridden by the heating plan rules, but as a programmer I might be more concerned about how the `targetTemperature` function has a dependency on a global thermostat object. I can break this dependency by moving it to a parameter.

My first step is to use *Extract Variable (119)* on the parameter that I want to have in my function.

*class HeatingPlan…*

```
get targetTemperature() {
  const selectedTemperature = thermostat.selectedTemperature;
  if      (selectedTemperature >  this._max) return this._max;
  else if (selectedTemperature <  this._min) return this._min;
  else return selectedTemperature;
}
```

That makes it easy to apply *Extract Function (106)* on the entire body of the function except for the bit that figures out the parameter.

*class HeatingPlan…*

```
get targetTemperature() {
  const selectedTemperature = thermostat.selectedTemperature;
  return this.xxNEWtargetTemperature(selectedTemperature);
}

xxNEWtargetTemperature(selectedTemperature) {
  if      (selectedTemperature >  this._max) return this._max;
  else if (selectedTemperature <  this._min) return this._min;
  else return selectedTemperature;
}
```

I then inline the variable I just extracted, which leaves the function as a simple call.

*class HeatingPlan…*

```
get targetTemperature() {
  return this.xxNEWtargetTemperature(thermostat.selectedTemperature);
}
```

I can now use *Inline Function (115)* on this method.

*caller…*

```
if      (thePlan.xxNEWtargetTemperature(thermostat.selectedTemperature) >
         thermostat.currentTemperature)
  setToHeat();
else if (thePlan.xxNEWtargetTemperature(thermostat.selectedTemperature) <
         thermostat.currentTemperature)
  setToCool();
else
  setOff();
```

I take advantage of the easily searchable name of the new function to rename it by removing the prefix.

*caller…*

```
if      (thePlan.targetTemperature(thermostat.selectedTemperature) >
         thermostat.currentTemperature)
  setToHeat();
else if (thePlan.targetTemperature(thermostat.selectedTemperature) <
         thermostat.currentTemperature)
  setToCool();
else
  setOff();
```

*class HeatingPlan…*

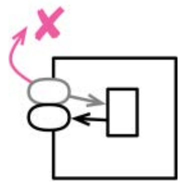
```
targetTemperature(selectedTemperature) {
  if      (selectedTemperature >  this._max) return this._max;
  else if (selectedTemperature <  this._min) return this._min;
  else return selectedTemperature;
}
```

As is often the case with this refactoring, the calling code looks more unwieldy than before. Moving a dependency out of a module pushes the responsibility of dealing with that dependency back to the caller. That's the trade-off for the reduced coupling.

But removing the coupling to the thermostat object isn't the only gain I've made with this refactoring. The `HeatingPlan` class is immutable—its fields are set in the constructor with no methods to alter them. (I'll save you the effort of looking at the whole class; just trust me on this.) Given an immutable heating plan, by moving the thermostat reference out of the function body I've also made `targetTemperature` referentially transparent. Every time I call `targetTemperature` on the same object, with the same argument, I will get the same result. If all the methods of the heating plan have referential transparency, that makes this class much easier to test and reason about.

A problem with JavaScript's class model is that it's impossible to enforce an immutable class—there's always a way to get at an object's data. But writing a class to signal and encourage immutability is often good enough. Creating classes that have this characteristic is often a sound strategy and Replace Query with Parameter is a handy tool for doing this.

## Remove Setting Method



```
class Person {
  get name() {...}
  set name(aString) {...}
```

⇓

```
class Person {
  get name() {...}
```

### Motivation

Providing a setting method indicates that a field may be changed. If I don't want that field to change once the object is created, I don't provide a setting method (and make the field immutable). That way, the field is set only in the constructor, my intention to have it not change is clear, and I usually remove the very possibility that the field will change.

There's a couple of common cases where this comes up. One is where people always use accessor methods to manipulate a field, even within constructors. This leads to the only call to a setting method being from the constructor. I prefer to remove the setting method to make it clear that updates make no sense after construction.

Another case is where the object is created by clients using creation script rather than by a simple constructor call. Such a creation script starts with the constructor call followed by a sequence of setter method calls to create the new object. Once the script is finished, we don't expect the new object to change some (or even all) of its fields. The setters are only expected to be called during this initial creation. In this case, I'd get rid of them to make my intentions clearer.

## Mechanics

- If the value that's being set isn't provided to the constructor, use *Change Function Declaration (124)* to add it. Add a call to the setting method within the constructor.

  If you wish to remove several setting methods, add all their values to the constructor at once. This simplifies the later steps.

- Remove each call of a setting method outside of the constructor, using the new constructor value instead. Test after each one.

  If you can't replace the call to the setter by creating a new object (because you are updating a shared reference object), abandon the refactoring.

- Use *Inline Function (115)* on the setting method. Make the field immutable if possible.

- Test.

## Example

I have a simple person class.

*class Person...*

```
get name()    {return this._name;}
set name(arg) {this._name = arg;}
get id()    {return this._id;}
set id(arg) {this._id = arg;}
```

At the moment, I create a new object with code like this:

```
const martin = new Person();
martin.name = "martin";
martin.id = "1234";
```

The name of a person may change after it's created, but the ID does not. To make this clear, I want to remove the setting method for ID.

I still need to set the ID initially, so I'll use *Change Function Declaration (124)* to add it to the constructor.

*class Person...*

```
constructor(id) {
  this.id = id;
}
```

I then adjust the creation script to set the ID via the constructor.

```
const martin = new Person("1234");
martin.name = "martin";
~~martin.id = "1234";~~
```

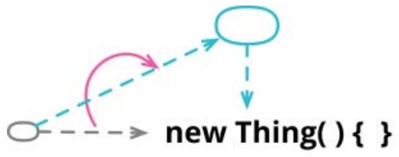
I do this in each place I create a person, testing after each change. When they are all done, I can apply *Inline Function (115)* to the setting method.

*class Person…*

```
constructor(id) {
  this._id = id;
}
get name()    {return this._name;}
set name(arg) {this._name = arg;}
get id()    {return this._id;}
~~set id(arg) {this._id = arg;}~~
```

## Replace Constructor with Factory Function

formerly: *Replace Constructor with Factory Method*



```
leadEngineer = new Employee(document.leadEngineer, 'E');
```

⇓

```
leadEngineer = createEngineer(document.leadEngineer);
```

### Motivation

Many object-oriented languages have a special constructor function that's called to initialize an object. Clients typically call this constructor when they want to create a new object. But these constructors often come with awkward limitations that aren't there for more general functions. A Java constructor must return an instance of the class it was called with, which means I can't replace it with a subclass or proxy depending on the environment or parameters. Constructor naming is fixed, which makes it impossible for me to use a name that is clearer than the default. Constructors often require a special operator to invoke ("new" in many languages) which makes them difficult to use in contexts that expect normal functions.

A factory function suffers from no such limitations. It will likely call the constructor as part of its implementation, but I can freely substitute something else.

### Mechanics

- Create a factory function, its body being a call to the constructor.
- Replace each call to the constructor with a call to the factory function.
- Test after each change.
- Limit the constructor's visibility as much as possible.

## Example

A quick but wearisome example uses kinds of employees. Consider an employee class:

*class Employee…*

```
constructor (name, typeCode) {
  this._name = name;
  this._typeCode = typeCode;
}
get name() {return this._name;}
get type() {
  return Employee.legalTypeCodes[this._typeCode];
}
static get legalTypeCodes() {
  return {"E": "Engineer", "M": "Manager", "S": "Salesman"};
}
```

This is used from

*caller…*

```
candidate = new Employee(document.name, document.empType);
```

and

*caller…*

```
const leadEngineer = new Employee(document.leadEngineer, 'E');
```

My first step is to create the factory function. Its body is a simple delegation to the constructor.

*top level…*

```
  function createEmployee(name, typeCode) {
    return new Employee(name, typeCode);
  }
```

I then find the callers of the constructor and change them, one at a time, to use the factory function instead.

The first one is obvious:

*caller…*

```
candidate = createEmployee(document.name, document.empType);
```

With the second case, I could use the new factory function like this:

*caller…*

```
const leadEngineer = createEmployee(document.leadEngineer, 'E');
```

But I don't like using the type code here—it's generally a bad smell to pass a code as a literal string. So I prefer to create a new factory function that embeds the kind of employee I want into its name.

*caller…*

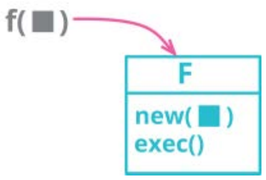
```
const leadEngineer = createEngineer(document.leadEngineer);
```

*top level…*

```
  function createEngineer(name) {
    return new Employee(name, 'E');
  }
```

## Replace Function with Command

formerly: *Replace Method with Method Object*
inverse of: *Replace Command with Function (344)*



```
function score(candidate, medicalExam, scoringGuide) {
  let result = 0;
  let healthLevel = 0;
  // long body code
}
```

⇓

```
class Scorer {
  constructor(candidate, medicalExam, scoringGuide) {
    this._candidate = candidate;
    this._medicalExam = medicalExam;
    this._scoringGuide = scoringGuide;
  }

  execute() {
    this._result = 0;
    this._healthLevel = 0;
    // long body code
  }
}
```

### Motivation

Functions—either freestanding or attached to objects as methods—are one of the fundamental building blocks of programming. But there are times when it's useful to encapsulate a function into its own object, which I refer to as a "command object" or simply a **command**. Such an object is mostly built around a single method, whose request and execution is the purpose of the object.

A command offers a greater flexibility for the control and expression of a function than the plain function mechanism. Commands can have complimentary operations, such as undo. I can provide methods to build up their parameters to support a richer lifecycle. I can build in customizations using inheritance and hooks. If I'm working in a language with objects but without first-class functions, I can provide much of that capability by using commands instead. Similarly, I can use methods and fields to help break down a complex function, even in a language that lacks nested functions, and I can call those methods directly while testing and debugging.

All these are good reasons to use commands, and I need to be ready to refactor functions into commands when I need to. But we must not forget that this flexibility, as ever, comes at a price paid in complexity. So, given the choice between a first-class function and a command, I'll pick the function 95% of the time. I only use a command when I specifically need a facility that simpler approaches can't provide.

Like many words in software development, "command" is rather overloaded. In the context I'm using it here, it is an object that encapsulates a request, following the command pattern in Design Patterns [gof]. When I use "command" in this sense, I use "command object" to set the context, and "command" afterwards. The word "command" is also used in the command-query separation principle [mf-cqs], where a command is an object method that changes observable state. I've always tried to avoid using command in that sense, preferring "modifier" or "mutator."

## Mechanics

- Create an empty class for the function. Name it based on the function.
- Use *Move Function (198)* to move the function to the empty class.

  Keep the original function as a forwarding function until at least the end of the refactoring.

  Follow any convention the language has for naming commands. If there is no convention, choose a generic name for the command's execute function, such as "execute" or "call".
- Consider making a field for each argument, and move these arguments to the constructor.

## Example

The JavaScript language has many faults, but one of its great decisions was to make functions first-class entities. I thus don't have to go through all the hoops of creating commands for common tasks that I need to do in languages without this facility. But there are still times when a command is the right tool for the job.

One of these cases is breaking up a complex function so I can better understand and modify it. To really show the value of this refactoring, I need a long and complicated function—but that would take too long to write, let alone for you to read. Instead, I'll go with a function that's short enough not to need it. This one scores points for an insurance application:

```
function score(candidate, medicalExam, scoringGuide) {
  let result = 0;
  let healthLevel = 0;
  let highMedicalRiskFlag = false;

  if (medicalExam.isSmoker) {
    healthLevel += 10;
    highMedicalRiskFlag = true;
  }
  let certificationGrade = "regular";
  if (scoringGuide.stateWithLowCertification(candidate.originState)) {
    certificationGrade = "low";
    result -= 5;
  }
  // lots more code like this
  result -= Math.max(healthLevel - 5, 0);
  return result;
}
```

I begin by creating an empty class and then *Move Function (198)* to move the function into it.

```
function score(candidate, medicalExam, scoringGuide) {
  return new Scorer().execute(candidate, medicalExam, scoringGuide);
}

 class Scorer {
   execute (candidate, medicalExam, scoringGuide) {
     let result = 0;
     let healthLevel = 0;
     let highMedicalRiskFlag = false;

     if (medicalExam.isSmoker) {
       healthLevel += 10;
       highMedicalRiskFlag = true;
     }
     let certificationGrade = "regular";
     if (scoringGuide.stateWithLowCertification(candidate.originState)) {
       certificationGrade = "low";
       result -= 5;
     }
     // lots more code like this
     result -= Math.max(healthLevel - 5, 0);
     return result;
   }
 }
```

Most of the time, I prefer to pass arguments to a command on the constructor and have the `execute` method take no parameters. While this matters less for a simple decomposition scenario like this, it's very handy when I want to manipulate the command with a more complicated parameter setting lifecycle or customizations. Different command classes can have different parameters but be mixed together when queued for execution.

I can do these parameters one at a time.

```
function score(candidate, medicalExam, scoringGuide) {
  return new Scorer(candidate).execute(~~candidate, ~~medicalExam, scoringGuide);
}
```

*class Scorer…*

```
  constructor(candidate){
    this._candidate = candidate;
  }

  execute (~~candidate, ~~medicalExam, scoringGuide) {
    let result = 0;
    let healthLevel = 0;
    let highMedicalRiskFlag = false;

    if (medicalExam.isSmoker) {
      healthLevel += 10;
      highMedicalRiskFlag = true;
    }
    let certificationGrade = "regular";
    if (scoringGuide.stateWithLowCertification(this._candidate.originState)) {
      certificationGrade = "low";
      result -= 5;
    }
    // lots more code like this
    result -= Math.max(healthLevel - 5, 0);
    return result;
  }
```

I continue with the other parameters

```
function score(candidate, medicalExam, scoringGuide) {
  return new Scorer(candidate, medicalExam, scoringGuide).execute();
}
```

*class Scorer…*

```
constructor(candidate, medicalExam, scoringGuide){
  this._candidate = candidate;
  this._medicalExam = medicalExam;
  this._scoringGuide = scoringGuide;
}
```

```
execute () {
  let result = 0;
  let healthLevel = 0;
  let highMedicalRiskFlag = false;

  if (this._medicalExam.isSmoker) {
    healthLevel += 10;
    highMedicalRiskFlag = true;
  }
  let certificationGrade = "regular";
  if (this._scoringGuide.stateWithLowCertification(this._candidate.originState)) {
    certificationGrade = "low";
    result -= 5;
  }
  // lots more code like this
  result -= Math.max(healthLevel - 5, 0);
  return result;
}
```

That completes Replace Function with Command, but the whole point of doing this refactoring is to allow me to break down the complicated functions—so let me outline some steps to achieve that. My next move here is to change all the local variables into fields. Again, I do these one at a time.

*class Scorer…*

```
constructor(candidate, medicalExam, scoringGuide){
  this._candidate = candidate;
  this._medicalExam = medicalExam;
  this._scoringGuide = scoringGuide;
}

execute () {
  this._result = 0;
  let healthLevel = 0;
  let highMedicalRiskFlag = false;

  if (this._medicalExam.isSmoker) {
    healthLevel += 10;
    highMedicalRiskFlag = true;
  }
  let certificationGrade = "regular";
  if (this._scoringGuide.stateWithLowCertification(this._candidate.originState)) {
    certificationGrade = "low";
    this._result -= 5;
  }
  // lots more code like this
  this._result -= Math.max(healthLevel - 5, 0);
  return this._result;
}
```

I repeat this for all the local variables. (This is one of those refactorings that I felt was sufficiently simple that I haven't given it an entry in the catalog. I feel slightly guilty about this.)

*class Scorer...*

```
constructor(candidate, medicalExam, scoringGuide){
  this._candidate = candidate;
  this._medicalExam = medicalExam;
  this._scoringGuide = scoringGuide;
}

execute () {
  this._result = 0;
  this._healthLevel = 0;
  this._highMedicalRiskFlag = false;

  if (this._medicalExam.isSmoker) {
    this._healthLevel += 10;
    this._highMedicalRiskFlag = true;
  }
  this._certificationGrade = "regular";
  if (this._scoringGuide.stateWithLowCertification(this._candidate.originState)) {
    this._certificationGrade = "low";
    this._result -= 5;
  }
  // lots more code like this
  this._result -= Math.max(this._healthLevel - 5, 0);
  return this._result;
}
```

Now I've moved all the function's state to the command object, I can use refactorings like *Extract Function (106)* without getting tangled up in all the variables and their scopes.

*class Scorer...*

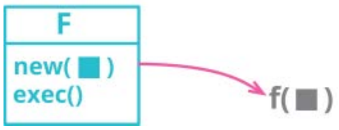
```
execute () {
  this._result = 0;
  this._healthLevel = 0;
  this._highMedicalRiskFlag = false;

  this.scoreSmoking();
  this._certificationGrade = "regular";
  if (this._scoringGuide.stateWithLowCertification(this._candidate.originState)) {
    this._certificationGrade = "low";
    this._result -= 5;
  }
  // lots more code like this
  this._result -= Math.max(this._healthLevel - 5, 0);
  return this._result;
  }
```

```
scoreSmoking() {
  if (this._medicalExam.isSmoker) {
    this._healthLevel += 10;
    this._highMedicalRiskFlag = true;
  }
}
```

This allows me to treat the command similarly to how I'd deal with a nested function. Indeed, when doing this refactoring in JavaScript, using nested functions would be a reasonable alternative to using a command. I'd still use a command for this, partly because I'm more familiar with commands and partly because with a command I can write tests and debugging calls against the subfunctions.

## Replace Command with Function

inverse of: *Replace Function with Command (337)*



```
class ChargeCalculator {
  constructor (customer, usage){
    this._customer = customer;
    this._usage = usage;
  }
  execute() {
    return this._customer.rate * this._usage;
  }
}
```

⇓

```
function charge(customer, usage) {
  return customer.rate * usage;
}
```

### Motivation

Command objects provide a powerful mechanism for handling complex computations. They can easily be broken down into separate methods sharing common state through the fields; they can be invoked via different methods for different effects; they can have their data built up in stages. But that power comes at a cost. Most of the time, I just want to invoke a function and have it do its thing. If that's the case, and the function isn't too complex, then a command object is more trouble than its worth and should be turned into a regular function.

### Mechanics

- Apply *Extract Function (106)* to the creation of the command and the call to the command's execution method.

  This creates the new function that will replace the command in due course.

- For each method called by the command's execution method, apply *Inline Function (115)*.

  If the supporting function returns a value, use *Extract Variable (119)* on the call first and then *Inline Function (115)*.

- Use *Change Function Declaration (124)* to put all the parameters of the constructor into the command's execution method instead.
- For each field, alter the references in the command's execution method to use the parameter instead. Test after each change.
- Inline the constructor call and command's execution method call into the caller (which is the replacement function).
- Test.
- Apply *Remove Dead Code (237)* to the command class.

## Example

I'll begin with this small command object:

```
class ChargeCalculator {
  constructor (customer, usage, provider){
    this._customer = customer;
    this._usage = usage;
    this._provider = provider;
  }
  get baseCharge() {
    return this._customer.baseRate * this._usage;
  }
  get charge() {
    return this.baseCharge + this._provider.connectionCharge;
  }
}
```

It is used by code like this:

*caller…*

```
monthCharge = new ChargeCalculator(customer, usage, provider).charge;
```

The command class is small and simple enough to be better off as a function. I begin by using *Extract Function (106)* to wrap the class creation and invocation.

*caller…*

```
monthCharge = charge(customer, usage, provider);
```

*top level…*

```
function charge(customer, usage, provider) {
  return new ChargeCalculator(customer, usage, provider).charge;
}
```

I have to decide how to deal with any supporting functions, in this case `baseCharge`. My usual approach for a function that returns a value is to first *Extract Variable (119)* on that value.

*class ChargeCalculator…*

```
get baseCharge() {
  return this._customer.baseRate * this._usage;
}
get charge() {
  const baseCharge = this.baseCharge;
  return baseCharge + this._provider.connectionCharge;
}
```

Then, I use *Inline Function (115)* on the supporting function.

*class ChargeCalculator…*

```
get charge() {
  const baseCharge = this._customer.baseRate * this._usage;
  return baseCharge + this._provider.connectionCharge;
}
```

I now have all the processing in a single function, so my next step is to move the data passed to the constructor to the main method. I first use *Change Function Declaration (124)* to add all the constructor parameters to the `charge` method.

*class ChargeCalculator…*

```
constructor (customer, usage, provider){
  this._customer = customer;
  this._usage = usage;
  this._provider = provider;
}

charge(customer, usage, provider) {
  const baseCharge = this._customer.baseRate * this._usage;
  return baseCharge + this._provider.connectionCharge;
}
```

*top level…*

```
function charge(customer, usage, provider) {
  return new ChargeCalculator(customer, usage, provider)
                    .charge(customer, usage, provider);
}
```

Now I can alter the body of `charge` to use the passed parameters instead. I can do this one at a time.

*class ChargeCalculator…*

```
constructor (customer, usage, provider){
  ~~this._customer = customer;~~
  this._usage = usage;
  this._provider = provider;
}

charge(customer, usage, provider) {
  const baseCharge = customer.baseRate * this._usage;
  return baseCharge + this._provider.connectionCharge;
}
```

I don't have to remove the assignment to `this._customer` in the constructor, as it will just be ignored. But I prefer to do it since that will make a test fail if I miss changing a use of field to the parameter. (And if a test doesn't fail, I should consider adding a new test.)

I repeat this for the other parameters, ending up with

*class ChargeCalculator…*

```
charge(customer, usage, provider) {
  const baseCharge = customer.baseRate * usage;
  return baseCharge + provider.connectionCharge;
}
```

Once I've done all of these, I can inline into the top-level `charge` function. This is a special kind of *Inline Function (115)*, as it's inlining both the constructor and method call together.

*top level…*

```
function charge(customer, usage, provider) {
    const baseCharge = customer.baseRate * usage;
    return baseCharge + provider.connectionCharge;
}
```

The command class is now dead code, so I'll use *Remove Dead Code (237)* to give it an honorable burial.

*This page intentionally left blank*

# Chapter 12

# Dealing with Inheritance

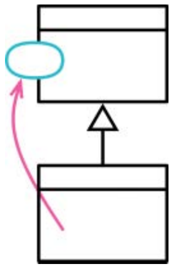
In this final chapter, I'll turn to one of the best known features of object-oriented programming: inheritance. Like any powerful mechanism, it is both very useful and easy to misuse, and it's often hard to see the misuse until it's in the rear-view mirror.

Often, features need to move up or down the inheritance hierarchy. Several refactorings deal with that: *Pull Up Method (350)*, *Pull Up Field (353)*, *Pull Up Constructor Body (355)*, *Push Down Method (359)*, and *Push Down Field (361)*. I can add and remove classes from the hierarchy with *Extract Superclass (375)*, *Remove Subclass (369)*, and *Collapse Hierarchy (380)*. I may want to add a subclass to replace a field that I'm using to trigger different behavior based on its value; I do this with *Replace Type Code with Subclasses (362)*.

Inheritance is a powerful tool, but sometimes it gets used in the wrong place—or the place it's used in becomes wrong. In that case, I use *Replace Subclass with Delegate (381)* or *Replace Superclass with Delegate (399)* to turn inheritance into delegation.

## Pull Up Method

inverse of: *Push Down Method (359)*



```
class Employee {...}

class Salesman extends Employee {
  get name() {...}
}

class Engineer extends Employee {
  get name() {...}
}
```

$\Downarrow$

```
class Employee {
  get name() {...}
}

class Salesman extends Employee {...}
class Engineer extends Employee {...}
```

### Motivation

Eliminating duplicate code is important. Two duplicate methods may work fine as they are, but they are nothing but a breeding ground for bugs in the future. Whenever there is duplication, there is risk that an alteration to one copy will not be made to the other. Usually, it is difficult to find the duplicates.

The easiest case of using Pull Up Method is when the methods have the same body, implying there's been a copy and paste. Of course it's not always as obvious as that. I could just do the refactoring and see if the tests croak—but that puts a lot of reliance on my tests. I usually find it valuable to look for the differences—often, they show up behavior that I forgot to test for.

Often, Pull Up Method comes after other steps. I see two methods in different classes that can be parameterized in such a way that they end up as essentially the same method. In that case, the smallest step is for me to apply *Parameterize Function (310)* separately and then Pull Up Method.

The most awkward complication with Pull Up Method is if the body of the method refers to features that are on the subclass but not on the superclass. When that happens, I need to use *Pull Up Field (353)* and Pull Up Method on those elements first.

If I have two methods with a similar overall flow, but differing in details, I'll consider the Form Template Method [mf-ft].

## Mechanics

- Inspect methods to ensure they are identical.

  If they do the same thing, but are not identical, refactor them until they have identical bodies.

- Check that all method calls and field references inside the method body refer to features that can be called from the superclass.
- If the methods have different signatures, use *Change Function Declaration (124)* to get them to the one you want to use on the superclass.
- Create a new method in the superclass. Copy the body of one of the methods over to it.
- Run static checks.
- Delete one subclass method.
- Test.
- Keep deleting subclass methods until they are all gone.

## Example

I have two subclass methods that do the same thing.

*class Employee extends Party…*

```
get annualCost() {
  return this.monthlyCost * 12;
}
```

*class Department extends Party…*

```
get totalAnnualCost() {
  return this.monthlyCost * 12;
}
```

I look at both classes and see that they refer to the `monthlyCost` property which isn't defined on the superclass, but is present in both subclasses. Since I'm in a dynamic language, I'm OK; if I were in a static language, I'd need to define an abstract method on `Party`.

The methods have different names, so I *Change Function Declaration (124)* to make them the same.

*class Department…*

```
get annualCost() {
  return this.monthlyCost * 12;
}
```

I copy the method from one subclass and paste it into the superclass.

*class Party…*

```
get annualCost() {
  return this.monthlyCost * 12;
}
```

In a static language, I'd compile to ensure that all the references were OK. That won't help me here, so I first remove `annualCost` from `Employee`, test, and then remove it from `Department`.

That completes the refactoring, but does leave a question. `annualCost` calls `monthlyCost`, but `monthlyCost` doesn't appear in the `Party` class. It all works, because JavaScript is a dynamic language—but there is value in signaling that subclasses of `Party` should provide an implementation for `monthlyCost`, particularly if more subclasses get added later on. A good way to provide this signal is a trap method like this:

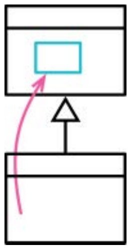
*class Party…*

```
get monthlyCost() {
  throw new SubclassResponsibilityError();
}
```

I call such an error a subclass responsibility error as that was the name used in Smalltalk.

## Pull Up Field

inverse of: *Push Down Field (361)*



```
class Employee {...} // Java

class Salesman extends Employee {
  private String name;
}

class Engineer extends Employee {
  private String name;
}
```

⇓

```
class Employee {
  protected String name;
}

class Salesman extends Employee {...}
class Engineer extends Employee {...}
```

### Motivation

If subclasses are developed independently, or combined through refactoring, I often find that they duplicate features. In particular, certain fields can be duplicates. Such fields sometimes have similar names—but not always. The only way I can tell what is going on is by looking at the fields and examining how they are used. If they are being used in a similar way, I can pull them up into the superclass.

By doing this, I reduce duplication in two ways. I remove the duplicate data declaration and I can then move behavior that uses the field from the subclasses to the superclass.

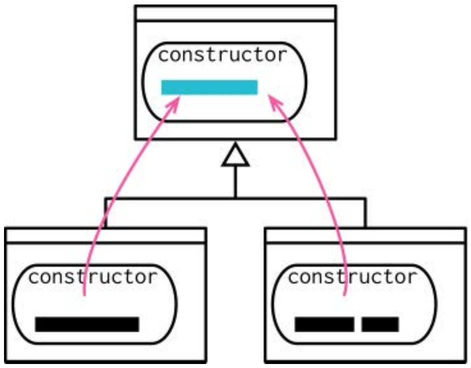
Many dynamic languages do not define fields as part of their class definition—instead, fields appear when they are first assigned to. In this case, pulling up a field is essentially a consequence of *Pull Up Constructor Body (355)*.

## Mechanics

- Inspect all users of the candidate field to ensure they are used in the same way.
- If the fields have different names, use *Rename Field (244)* to give them the same name.
- Create a new field in the superclass.

  The new field will need to be accessible to subclasses (`protected` in common languages).
- Delete the subclass fields.
- Test.

## Pull Up Constructor Body



```
class Party {...}

class Employee extends Party {
  constructor(name, id, monthlyCost) {
    super();
    this._id = id;
    this._name = name;
    this._monthlyCost = monthlyCost;
  }
}
```

⇓

```
class Party {
  constructor(name){
    this._name = name;
  }
}

class Employee extends Party {
  constructor(name, id, monthlyCost) {
    super(name);
    this._id = id;
    this._monthlyCost = monthlyCost;
  }
}
```

## Motivation

Constructors are tricky things. They aren't quite normal methods—so I'm more restricted in what I can do with them.

If I see subclass methods with common behavior, my first thought is to use *Extract Function (106)* followed by *Pull Up Method (350)*, which will move it nicely into the superclass. Constructors tangle that—because they have special rules about what can be done in what order, so I need a slightly different approach.

If this refactoring starts getting messy, I reach for *Replace Constructor with Factory Function (334)*.

## Mechanics

- Define a superclass constructor, if one doesn't already exist. Ensure it's called by subclass constructors.
- Use *Slide Statements (223)* to move any common statements to just after the super call.
- Remove the common code from each subclass and put it in the superclass. Add to the super call any constructor parameters referenced in the common code.
- Test.
- If there is any common code that cannot move to the start of the constructor, use *Extract Function (106)* followed by *Pull Up Method (350)*.

## Example

I start with the following code:

```
class Party {}

class Employee extends Party {
  constructor(name, id, monthlyCost) {
    super();
    this._id = id;
    this._name = name;
    this._monthlyCost = monthlyCost;
  }
  // rest of class...
```

```
class Department extends Party {
  constructor(name, staff){
    super();
    this._name = name;
    this._staff = staff;
  }
  // rest of class...
```

The common code here is the assignment of the name. I use *Slide Statements (223)* to move the assignment in `Employee` next to the call to `super()`:

```
class Employee extends Party {
  constructor(name, id, monthlyCost) {
    super();
    this._name = name;
    this._id = id;
    this._monthlyCost = monthlyCost;
  }
  // rest of class...
```

With that tested, I move the common code to the superclass. Since that code contains a reference to a constructor argument, I pass that in as a parameter.

*class Party...*

```
 constructor(name){
   this._name = name;
 }
```

*class Employee...*

```
 constructor(name, id, monthlyCost) {
   super(name);
   this._id = id;
   this._monthlyCost = monthlyCost;
 }
```

*class Department...*

```
 constructor(name, staff){
   super(name);
   this._staff = staff;
 }
```

Run the tests, and I'm done.

Most of the time, constructor behavior will work like this: Do the common elements first (with a `super` call), then do extra work that the subclass needs. Occasionally, however, there is some common behavior later.

Consider this example:

*class Employee...*

```
constructor (name) {...}

get isPrivileged() {...}

assignCar() {...}
```

*class Manager extends Employee...*

```
constructor(name, grade) {
  super(name);
  this._grade = grade;
  if (this.isPrivileged) this.assignCar(); // every subclass does this
}

get isPrivileged() {
  return this._grade > 4;
}
```

The wrinkle here comes from the fact that the call to `isPrivileged` can't be made until after the `grade` field is assigned, and that can only be done in the subclass.

In this case, I do *Extract Function (106)* on the common code:

*class Manager...*

```
constructor(name, grade) {
  super(name);
  this._grade = grade;
  this.finishConstruction();
}

finishConstruction() {
  if (this.isPrivileged) this.assignCar();
}
```

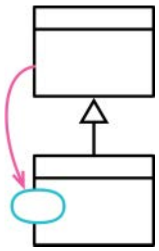
Then, I use *Pull Up Method (350)* to move it to the superclass.

*class Employee...*

```
finishConstruction() {
  if (this.isPrivileged) this.assignCar();
}
```

## Push Down Method

inverse of: *Pull Up Method (350)*



```
class Employee {
  get quota {...}
}

class Engineer extends Employee {...}
class Salesman extends Employee {...}
```

⇓

```
class Employee {...}
class Engineer extends Employee {...}
class Salesman extends Employee {
  get quota {...}
}
```

### Motivation

If a method is only relevant to one subclass (or a small proportion of subclasses), removing it from the superclass and putting it only on the subclass(es) makes that clearer. I can only do this refactoring if the caller knows it's working with a particular subclass—otherwise, I should use *Replace Conditional with Polymorphism (272)* with some placebo behavior on the superclass.

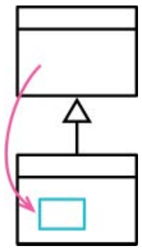
### Mechanics

- Copy the method into every subclass that needs it.
- Remove the method from the superclass.

- Test.
- Remove the method from each superclass that doesn't need it.
- Test.

## Push Down Field

inverse of: *Pull Up Field (353)*



```
class Employee {        // Java
  private String quota;
}

class Engineer extends Employee {...}
class Salesman extends Employee {...}
```

⇓

```
class Employee {...}
class Engineer extends Employee {...}

class Salesman extends Employee {
  protected String quota;
}
```

### Motivation

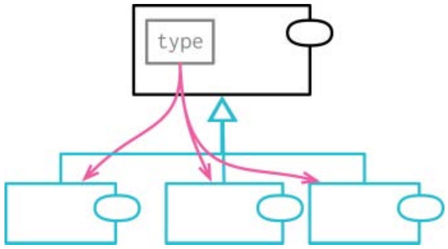
If a field is only used by one subclass (or a small proportion of subclasses), I move it to those subclasses.

### Mechanics

- Declare field in all subclasses that need it.
- Remove the field from the superclass.
- Test.
- Remove the field from all subclasses that don't need it.
- Test.

## Replace Type Code with Subclasses

subsumes: *Replace Type Code with State/Strategy*
subsumes: *Extract Subclass*
inverse of: *Remove Subclass (369)*



```
function createEmployee(name, type) {
  return new Employee(name, type);
}
```

⇓

```
function createEmployee(name, type) {
  switch (type) {
    case "engineer": return new Engineer(name);
    case "salesman": return new Salesman(name);
    case "manager":  return new Manager (name);
  }
```

### Motivation

Software systems often need to represent different kinds of a similar thing. I may classify employees by their job type (engineer, manager, salesman), or orders by their priority (rush, regular). My first tool for handling this is some kind of type code field—depending on the language, that might be an enum, symbol, string, or number. Often, this type code will come from an external service that provides me with the data I'm working on.

Most of the time, such a type code is all I need. But there are a couple of situations where I could do with something more, and that something more are subclasses. There are two things that are particularly enticing about subclasses. First, they allow me to use polymorphism to handle conditional logic. I find this most helpful when I have several functions that invoke different behavior depending on the value of the type code. With subclasses, I can apply *Replace Conditional with Polymorphism (272)* to these functions.

The second case is where I have fields or methods that are only valid for particular values of a type code, such as a sales quota that's only applicable to the "salesman" type code. I can then create the subclass and apply *Push Down Field (361)*. While I can include validation logic to ensure a field is only used when the type code has the correct value, using a subclass makes the relationship more explicit.

When using Replace Type Code with Subclasses, I need to consider whether to apply it directly to the class I'm looking at, or to the type code itself. Do I make engineer a subtype of employee, or should I give the employee an employee type property which can have subtypes for engineer and manager? Using direct subclassing is simpler, but I can't use it for the job type if I need it for something else. I also can't use direct subclasses if the type is mutable. If I need to move the subclasses to an employee type property, I can do that by using *Replace Primitive with Object (174)* on the type code to create an employee type class and then using Replace Type Code with Subclasses on that new class.

## Mechanics

- Self-encapsulate the type code field.
- Pick one type code value. Create a subclass for that type code. Override the type code getter to return the literal type code value.
- Create selector logic to map from the type code parameter to the new subclass.

  With direct inheritance, use *Replace Constructor with Factory Function (334)* and put the selector logic in the factory. With indirect inheritance, the selector logic may stay in the constructor.
- Test.
- Repeat creating the subclass and adding to the selector logic for each type code value. Test after each change.
- Remove the type code field.
- Test.
- Use *Push Down Method (359)* and *Replace Conditional with Polymorphism (272)* on any methods that use the type code accessors. Once all are replaced, you can remove the type code accessors.

## Example

I'll start with this overused employee example:

*class Employee...*

```
constructor(name, type){
  this.validateType(type);
  this._name = name;
  this._type = type;
}
validateType(arg) {
  if (!["engineer", "manager", "salesman"].includes(arg))
    throw new Error(`Employee cannot be of type ${arg}`);
}
toString() {return `${this._name} (${this._type})`;}
```

My first step is to use *Encapsulate Variable (132)* to self-encapsulate the type code.

*class Employee...*

```
  get type() {return this._type;}
  toString() {return `${this._name} (${this.type})`;}
```

*Note that* `toString` *uses the new getter by removing the underscore.*

I pick one type code, the engineer, to start with. I use direct inheritance, subclassing the employee class itself. The employee subclass is simple—just overriding the type code getter with the appropriate literal value.

```
class Engineer extends Employee {
   get type() {return "engineer";}
}
```

Although JavaScript constructors can return other objects, things will get messy if I try to put selector logic in there, since that logic gets intertwined with field initialization. So I use *Replace Constructor with Factory Function (334)* to create a new space for it.

```
function createEmployee(name, type) {
  return new Employee(name, type);
}
```

To use the new subclass, I add selector logic into the factory.

```
function createEmployee(name, type) {
  switch (type) {
    case "engineer": return new Engineer(name, type);
  }
  return new Employee(name, type);
}
```

I test to ensure that worked out correctly. But, because I'm paranoid, I then alter the return value of the engineer's override and test again to ensure the test fails. That way I know the subclass is being used. I correct the return value and continue with the other cases. I can do them one at a time, testing after each change.

```
class Salesman extends Employee {
   get type() {return "salesman";}
}

class Manager extends Employee {
   get type() {return "manager";}
}

function createEmployee(name, type) {
  switch (type) {
    case "engineer": return new Engineer(name, type);
    case "salesman": return new Salesman(name, type);
    case "manager":  return new Manager (name, type);
  }
  return new Employee(name, type);
}
```

Once I'm done with them all, I can remove the type code field and the superclass getting method (the ones in the subclasses remain).

*class Employee...*

```
constructor(name, type){
  this.validateType(type);
  this._name = name;
  ~~this._type = type;~~
}

~~get type() {return this._type;}~~
toString() {return `${this._name} (${this.type})`;}
```

After testing to ensure all is still well, I can remove the validation logic, since the switch is effectively doing the same thing.

*class Employee...*

```
constructor(name, type){
  ~~this.validateType(type);~~
  this._name = name;
}
```

```
function createEmployee(name, type) {
  switch (type) {
    case "engineer": return new Engineer(name, type);
    case "salesman": return new Salesman(name, type);
    case "manager":  return new Manager (name, type);
    default: throw new Error(`Employee cannot be of type ${type}`);
  }
  ~~return new Employee(name, type);~~
}
```

The type argument to the constructor is now useless, so it falls victim to *Change Function Declaration (124)*.

*class Employee…*

```
 constructor(name~~, type~~){
   this._name = name;
 }

function createEmployee(name, type) {
  switch (type) {
    case "engineer": return new Engineer(name~~, type~~);
    case "salesman": return new Salesman(name~~, type~~);
    case "manager":  return new Manager (name~~, type~~);
    default: throw new Error(`Employee cannot be of type ${type}`);
  }
}
```

I still have the type code accessors on the subclasses—`get type`. I'll usually want to remove these too, but that may take a bit of time due to other methods that depend on them. I'll use *Replace Conditional with Polymorphism (272)* and *Push Down Method (359)* to deal with these. At some point, I'll have no code that uses the `type` getters, so I will subject them to the tender mercies of *Remove Dead Code (237)*.

## Example: Using Indirect Inheritance

Let's go back to the starting case—but this time, I already have existing subclasses for part-time and full-time employees, so I can't subclass from `Employee` for the type codes. Another reason to not use direct inheritance is keeping the ability to change the type of employee.

*class Employee...*

```
constructor(name, type){
  this.validateType(type);
  this._name = name;
  this._type = type;
}
validateType(arg) {
  if (!["engineer", "manager", "salesman"].includes(arg))
    throw new Error(`Employee cannot be of type ${arg}`);
}
get type()    {return this._type;}
set type(arg) {this._type = arg;}

get capitalizedType() {
  return this._type.charAt(0).toUpperCase() + this._type.substr(1).toLowerCase();
}
toString() {
  return `${this._name} (${this.capitalizedType})`;
}
```

This time toString is a bit more complicated, to allow me to illustrate something shortly.

My first step is to use *Replace Primitive with Object (174)* on the type code.

```
class EmployeeType {
  constructor(aString) {
    this._value = aString;
  }
  toString() {return this._value;}
}
```

*class Employee...*

```
constructor(name, type){
  this.validateType(type);
  this._name = name;
  this.type = type;
}
validateType(arg) {
  if (!["engineer", "manager", "salesman"].includes(arg))
    throw new Error(`Employee cannot be of type ${arg}`);
}
get typeString()    {return this._type.toString();}
get type()    {return this._type;}
set type(arg) {this._type = new EmployeeType(arg);}

get capitalizedType() {
  return this.typeString.charAt(0).toUpperCase()
    + this.typeString.substr(1).toLowerCase();
}
```

```
toString() {
  return `${this._name} (${this.capitalizedType})`;
}
```

I then apply the usual mechanics of Replace Type Code with Subclasses to the employee type.

*class Employee…*

```
set type(arg) {this._type = Employee.createEmployeeType(arg);}

  static createEmployeeType(aString) {
    switch(aString) {
      case "engineer": return new Engineer();
      case "manager":  return new Manager ();
      case "salesman": return new Salesman();
      default: throw new Error(`Employee cannot be of type ${aString}`);
    }
  }

class EmployeeType {
}
class Engineer extends EmployeeType {
  toString() {return "engineer";}
}
class Manager extends EmployeeType {
  toString() {return "manager";}
}
class Salesman extends EmployeeType {
  toString() {return "salesman";}
}
```

If I were leaving it at that, I could remove the empty `EmployeeType`. But I prefer to leave it there as it makes explicit the relationship between the various subclasses. It's also a handy spot for moving other behavior there, such as the capitalization logic I tossed into the example specifically to illustrate this point.

*class Employee…*

```
toString() {
  return `${this._name} (${this.type.capitalizedName})`;
}
```

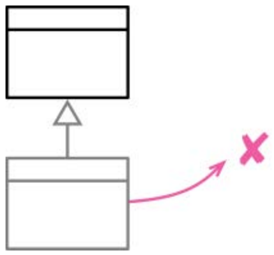
*class EmployeeType…*

```
  get capitalizedName() {
    return this.toString().charAt(0).toUpperCase()
      + this.toString().substr(1).toLowerCase();
  }
```

For those familiar with the first edition of the book, this example essentially supersedes the Replace Type Code with State/Strategy. I now think of that refactoring as Replace Type Code with Subclasses using indirect inheritance, so didn't consider it worth its own entry in the catalog. (I never liked the name anyway.)

## Remove Subclass

formerly: *Replace Subclass with Fields*
inverse of: *Replace Type Code with Subclasses (362)*



```
class Person {
  get genderCode() {return "X";}
}
class Male extends Person {
  get genderCode() {return "M";}
}
class Female extends Person {
  get genderCode() {return "F";}
}
```

⇓

```
class Person {
  get genderCode() {return this._genderCode;}
}
```

### Motivation

Subclasses are useful. They support variations in data structure and polymorphic behavior. They are a good way to program by difference. But as a software system evolves, subclasses can lose their value as the variations they support are moved to other places or removed altogether. Sometimes, subclasses are added in anticipation of features that never end up being built, or end up being built in a way that doesn't need the subclasses.

A subclass that does too little incurs a cost in understanding that is no longer worthwhile. When that time comes, it's best to remove the subclass, replacing it with a field on its superclass.

## Mechanics

- Use *Replace Constructor with Factory Function (334)* on the subclass constructor.

  If the clients of the constructors use a data field to decide which subclass to create, put that decision logic into a superclass factory method.

- If any code tests against the subclass's types, use *Extract Function (106)* on the type test and *Move Function (198)* to move it to the superclass. Test after each change.
- Create a field to represent the subclass type.
- Change the methods that refer to the subclass to use the new type field.
- Delete the subclass.
- Test.

Often, this refactoring is used on a group of subclasses at once—in which case carry out the steps to encapsulate them (add factory function, move type tests) first, then individually fold them into the superclass.

## Example

I'll start with this stump of subclasses:

*class Person…*

```
constructor(name) {
  this._name = name;
}
get name()    {return this._name;}
get genderCode() {return "X";}
// snip

class Male extends Person {
  get genderCode() {return "M";}
}

class Female extends Person {
  get genderCode() {return "F";}
}
```

If that's all that a subclass does, it's not really worth having. But before I remove these subclasses, it's usually worth checking to see if there's any subclass-dependent behavior in the clients that should be moved in there. In this case, I don't find anything worth keeping the subclasses for.

*client…*

```
const numberOfMales = people.filter(p => p instanceof Male).length;
```

Whenever I want to change how I represent something, I try to first encapsulate the current representation to minimize the impact on any client code. When it comes to creating subclasses, the way to encapsulate is to use *Replace Constructor with Factory Function (334)*. In this case, there's a couple of ways I could make the factory.

The most direct way is to create a factory method for each constructor.

```
function createPerson(name) {
  return new Person(name);
}
function createMale(name) {
  return new Male(name);
}
function createFemale(name) {
  return new Female(name);
}
```

But although that's the direct choice, objects like this are often loaded from a source that uses the gender codes directly.

```
function loadFromInput(data) {
  const result = [];
  data.forEach(aRecord => {
    let p;
    switch (aRecord.gender) {
      case 'M': p = new Male(aRecord.name); break;
      case 'F': p = new Female(aRecord.name); break;
      default: p = new Person(aRecord.name);
    }
    result.push(p);
  });
  return result;
}
```

In that case, I find it better to use *Extract Function (106)* on the selection logic for which class to create, and make that the factory function.

```
function createPerson(aRecord) {
  let p;
  switch (aRecord.gender) {
    case 'M': p = new Male(aRecord.name); break;
    case 'F': p = new Female(aRecord.name); break;
    default: p = new Person(aRecord.name);
  }
  return p;
}
```

```
function loadFromInput(data) {
  const result = [];
  data.forEach(aRecord => {
    result.push(createPerson(aRecord));
  });
  return result;
}
```

While I'm there, I'll clean up those two functions. I'll use *Inline Variable (123)* on `createPerson`:

```
function createPerson(aRecord) {
  switch (aRecord.gender) {
    case 'M': return new Male  (aRecord.name);
    case 'F': return new Female(aRecord.name);
    default:  return new Person(aRecord.name);
  }
}
```

and *Replace Loop with Pipeline (231)* on `loadFromInput`:

```
function loadFromInput(data) {
  return data.map(aRecord => createPerson(aRecord));
}
```

The factory encapsulates the creation of the subclasses, but there is also the use of `instanceof`—which never smells good. I use *Extract Function (106)* on the type check.

*client…*

```
 const numberOfMales = people.filter(p => isMale(p)).length;

function isMale(aPerson) {return aPerson instanceof Male;}
```

Then I use *Move Function (198)* to move it into `Person`.

*class Person…*

```
  get isMale() {return this instanceof Male;}
```

*client…*

```
 const numberOfMales = people.filter(p => p.isMale).length;
```

With that refactoring done, all knowledge of the subclasses is now safely encased within the superclass and the factory function. (Usually I'm wary of a superclass referring to a subclass, but this code isn't going to last until my next cup of tea, so I'm not going worry about it.)

I now add a field to represent the difference between the subclasses; since I'm using a code loaded from elsewhere, I might as well just use that.

*class Person...*

```
constructor(name, genderCode) {
  this._name = name;
  this._genderCode = genderCode || "X";
}

get genderCode() {return this._genderCode;}
```

When initializing it, I set it to the default case. (As a side note, although most people can be classified as male or female, there are people who can't. It's a common modeling mistake to forget that.)

I then take the male case and fold its logic into the superclass. This involves modifying the factory to return a `Person` and modifying any `instanceof` tests to use the gender code field.

```
function createPerson(aRecord) {
  switch (aRecord.gender) {
    case 'M': return new Person(aRecord.name, "M");
    case 'F': return new Female(aRecord.name);
    default:  return new Person(aRecord.name);
  }
}
```

*class Person...*

```
get isMale() {return "M" === this._genderCode;}
```

I test, remove the male subclass, test again, and repeat for the female subclass.

```
function createPerson(aRecord) {
  switch (aRecord.gender) {
    case 'M': return new Person(aRecord.name, "M");
    case 'F': return new Person(aRecord.name, "F");
    default:  return new Person(aRecord.name);
  }
}
```

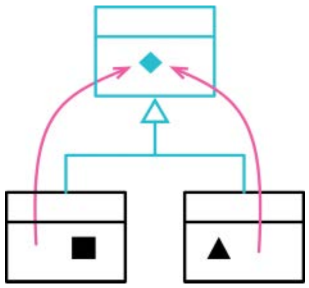
I find the lack of symmetry with the gender code to be annoying. A future reader of the code will always wonder about this lack of symmetry. So I prefer to change the code to make it symmetrical—if I can do it without introducing any other complexity, which is the case here.

```
function createPerson(aRecord) {
  switch (aRecord.gender) {
    case 'M': return new Person(aRecord.name, "M");
    case 'F': return new Person(aRecord.name, "F");
    default:  return new Person(aRecord.name, "X");
  }
}
```

*class Person…*

```
  constructor(name, genderCode) {
    this._name = name;
    this._genderCode = genderCode~~ || "X"~~;
  }
```

## Extract Superclass



```
class Department {
  get totalAnnualCost() {...}
  get name() {...}
  get headCount() {...}
}

class Employee {
  get annualCost() {...}
  get name() {...}
  get id() {...}
}
```

$\Downarrow$

```
class Party {
  get name() {...}
  get annualCost() {...}
}

class Department extends Party {
  get annualCost() {...}
  get headCount() {...}
}

class Employee extends Party {
  get annualCost() {...}
  get id() {...}
}
```

## Motivation

If I see two classes doing similar things, I can take advantage of the basic mechanism of inheritance to pull their similarities together into a superclass. I can use *Pull Up Field (353)* to move common data into the superclass, and *Pull Up Method (350)* to move the common behavior.

Many writers on object orientation treat inheritance as something that should be carefully planned in advance, based on some kind of classification structure in the "real world." Such classification structures can be a hint towards using inheritance—but just as often inheritance is something I realize during the evolution of a program, as I find common elements that I want to pull together.

An alternative to Extract Superclass is *Extract Class (182)*. Here you have, essentially, a choice between using inheritance or delegation as a way to unify duplicate behavior. Often Extract Superclass is the simpler approach, so I'll do this first knowing I can use *Replace Superclass with Delegate (399)* should I need to later.

## Mechanics

- Create an empty superclass. Make the original classes its subclasses.

  If needed, use *Change Function Declaration (124)* on the constructors.

- Test.

- One by one, use *Pull Up Constructor Body (355)*, *Pull Up Method (350)*, and *Pull Up Field (353)* to move common elements to the superclass.

- Examine remaining methods on the subclasses. See if there are common parts. If so, use *Extract Function (106)* followed by *Pull Up Method (350)*.

- Check clients of the original classes. Consider adjusting them to use the superclass interface.

## Example

I'm pondering these two classes, they share some common functionality—their name and the notions of annual and monthly costs:

```
class Employee {
  constructor(name, id, monthlyCost) {
    this._id = id;
    this._name = name;
    this._monthlyCost = monthlyCost;
  }
  get monthlyCost() {return this._monthlyCost;}
  get name() {return this._name;}
  get id() {return this._id;}

  get annualCost() {
    return this.monthlyCost * 12;
  }
}

class Department {
  constructor(name, staff){
    this._name = name;
    this._staff = staff;
  }
  get staff() {return this._staff.slice();}
  get name() {return this._name;}

  get totalMonthlyCost() {
    return this.staff
      .map(e => e.monthlyCost)
      .reduce((sum, cost) => sum + cost);
  }
  get headCount() {
    return this.staff.length;
  }
  get totalAnnualCost() {
    return this.totalMonthlyCost * 12;
  }
}
```

I can make the common behavior more explicit by extracting a common superclass from them.

I begin by creating an empty superclass and letting them both extend from it.

```
class Party {}

class Employee extends Party {
  constructor(name, id, monthlyCost) {
    super();
    this._id = id;
    this._name = name;
    this._monthlyCost = monthlyCost;
  }
  // rest of class...
```

```
class Department extends Party {
  constructor(name, staff){
    super();
    this._name = name;
    this._staff = staff;
  }
  // rest of class...
```

When doing Extract Superclass, I like to start with the data, which in JavaScript involves manipulating the constructor. So I start with *Pull Up Field (353)* to pull up the name.

*class Party…*

```
constructor(name){
  this._name = name;
}
```

*class Employee…*

```
constructor(name, id, monthlyCost) {
  super(name);
  this._id = id;
  this._monthlyCost = monthlyCost;
}
```

*class Department…*

```
constructor(name, staff){
  super(name);
  this._staff = staff;
}
```

As I get data up to the superclass, I can also apply *Pull Up Method (350)* on associated methods. First, the name:

*class Party…*

```
get name() {return this._name;}
```

*class Employee…*

~~get name() {return this._name;}~~

*class Department…*

~~get name() {return this._name;}~~

I have two methods with similar bodies.

*class Employee…*

```
get annualCost() {
  return this.monthlyCost * 12;
}
```

*class Department…*

```
get totalAnnualCost() {
  return this.totalMonthlyCost * 12;
}
```

The methods they use, `monthlyCost` and `totalMonthlyCost`, have different names and different bodies—but do they represent the same intent? If so, I should use *Change Function Declaration (124)* to unify their names.

*class Department…*

```
get totalAnnualCost() {
  return this.monthlyCost * 12;
}

get monthlyCost() { … }
```

I then do a similar renaming to the annual costs:

*class Department…*

```
get annualCost() {
  return this.monthlyCost * 12;
}
```

I can now apply *Pull Up Method (350)* to the annual cost methods.

*class Party…*

```
get annualCost() {
  return this.monthlyCost * 12;
}
```

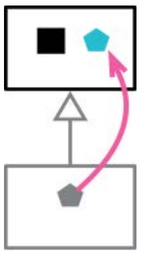
*class Employee…*

~~get annualCost() {~~
  ~~return this.monthlyCost * 12;~~
~~}~~

*class Department…*

~~get annualCost() {~~
  ~~return this.monthlyCost * 12;~~
~~}~~

## Collapse Hierarchy



```
class Employee {...}
class Salesman extends Employee {...}
```

⇓

```
class Employee {...}
```

### Motivation

When I'm refactoring a class hierarchy, I'm often pulling and pushing features around. As the hierarchy evolves, I sometimes find that a class and its parent are no longer different enough to be worth keeping separate. At this point, I'll merge them together.

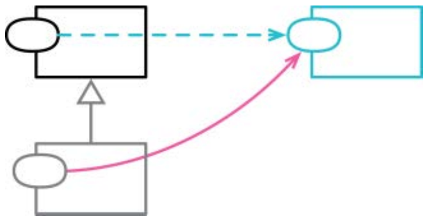
### Mechanics

- Choose which one to remove.

  I choose based on which name makes most sense in the future. If neither name is best, I'll pick one arbitrarily.

- Use *Pull Up Field (353)*, *Push Down Field (361)*, *Pull Up Method (350)*, and *Push Down Method (359)* to move all the elements into a single class.
- Adjust any references to the victim to change them to the class that will stay.
- Remove the empty class.
- Test.

## Replace Subclass with Delegate



```
class Order {
  get daysToShip() {
    return this._warehouse.daysToShip;
  }
}

class PriorityOrder extends Order {
  get daysToShip() {
    return this._priorityPlan.daysToShip;
  }
}
```

⇓

```
class Order {
  get daysToShip() {
    return (this._priorityDelegate)
      ? this._priorityDelegate.daysToShip
      : this._warehouse.daysToShip;
  }
}

class PriorityOrderDelegate {
  get daysToShip() {
    return this._priorityPlan.daysToShip
  }
}
```

### Motivation

If I have some objects whose behavior varies from category to category, the natural mechanism to express this is inheritance. I put all the common data and behavior in the superclass, and let each subclass add and override features as needed. Object-oriented languages make this simple to implement and thus a familiar mechanism.

But inheritance has its downsides. Most obviously, it's a card that can only be played once. If I have more than one reason to vary something, I can only use inheritance for a single axis of variation. So, if I want to vary behavior of people by their age category and by their income level, I can either have subclasses for young and senior, or for well-off and poor—I can't have both.

A further problem is that inheritance introduces a very close relationship between classes. Any change I want to make to the parent can easily break children, so I have to be careful and understand how children derive from the superclass. This problem is made worse when the logic of the two classes resides in different modules and is looked after by different teams.

Delegation handles both of these problems. I can delegate to many different classes for different reasons. Delegation is a regular relationship between objects—so I can have a clear interface to work with, which is much less coupling than subclassing. It's therefore common to run into the problems with subclassing and apply Replace Subclass with Delegate.

There is a popular principle: "Favor object composition over class inheritance" (where composition is effectively the same as delegation). Many people take this to mean "inheritance considered harmful" and claim that we should never use inheritance. I use inheritance frequently, partly because I always know I can use Replace Subclass with Delegate should I need to change it later. Inheritance is a valuable mechanism that does the job most of the time without problems. So I reach for it first, and move onto delegation when it starts to rub badly. This usage is actually consistent with the principle—which comes from the Gang of Four book [gof] that explains how inheritance and composition work together. The principle was a reaction to the overuse of inheritance.

Those who are familiar with the Gang of Four book may find it helpful to think of this refactoring as replacing subclasses with the State or Strategy patterns. Both of these patterns are structurally the same, relying on the host delegating to a separate hierarchy. Not all cases of Replace Subclass with Delegate involve an inheritance hierarchy for the delegate (as the first example below illustrates), but setting up a hierarchy for states or strategies is often useful.

## Mechanics

- If there are many callers for the constructors, apply *Replace Constructor with Factory Function (334)*.
- Create an empty class for the delegate. Its constructor should take any subclass-specific data as well as, usually, a back-reference to the superclass.
- Add a field to the superclass to hold the delegate.

- Modify the creation of the subclass so that it initializes the delegate field with an instance of the delegate.

  This can be done in the factory function, or in the constructor if the constructor can reliably tell whether to create the correct delegate.

- Choose a subclass method to move to the delegate class.
- Use *Move Function (198)* to move it to the delegate class. Don't remove the source's delegating code.

  If the method needs elements that should move to the delegate, move them. If it needs elements that should stay in the superclass, add a field to the delegate that refers to the superclass.

- If the source method has callers outside the class, move the source's delegating code from the subclass to the superclass, guarding it with a check for the presence of the delegate. If not, apply *Remove Dead Code (237)*.

  If there's more than one subclass, and you start duplicating code within them, use *Extract Superclass (375)*. In this case, any delegating methods on the source superclass no longer need a guard if the default behavior is moved to the delegate superclass.

- Test.
- Repeat until all the methods of the subclass are moved.
- Find all callers of the subclasses's constructor and change them to use the superclass constructor.
- Test.
- Use *Remove Dead Code (237)* on the subclass.

## Example

I have a class that makes a booking for a show.

*class Booking…*
```
  constructor(show, date) {
    this._show = show;
    this._date = date;
  }
```

There is a subclass for premium booking that takes into account various extras that are available.

*class PremiumBooking extends Booking…*

```
constructor(show, date, extras) {
  super(show, date);
  this._extras = extras;
}
```

There are quite a few changes that the premium booking makes to what it inherits from the superclass. As is typical with this kind of programming-by-difference, in some cases the subclass overrides methods on the superclass, in others it adds new methods that are only relevant for the subclass. I won't go into all of them, but I will pick out a few interesting cases.

First, there is a simple override. Regular bookings offer a talkback after the show, but only on nonpeak days.

*class Booking…*

```
get hasTalkback() {
  return this._show.hasOwnProperty('talkback') && !this.isPeakDay;
}
```

Premium bookings override this to offer talkbacks on all days.

*class PremiumBooking…*

```
get hasTalkback() {
  return this._show.hasOwnProperty('talkback');
}
```

Determining the price is a similar override, with a twist that the premium method calls the superclass method.

*class Booking…*

```
get basePrice() {
  let result = this._show.price;
  if (this.isPeakDay) result += Math.round(result * 0.15);
  return result;
}
```

*class PremiumBooking…*

```
get basePrice() {
  return Math.round(super.basePrice + this._extras.premiumFee);
}
```

The last example is where the premium booking offers a behavior that isn't present on the superclass.

*class PremiumBooking…*

```
get hasDinner() {
  return this._extras.hasOwnProperty('dinner') && !this.isPeakDay;
}
```

Inheritance works well for this example. I can understand the base class without having to understand the subclass. The subclass is defined just by saying how it differs from the base case—both reducing duplication and clearly communicating what are the differences it's introducing.

Actually, it isn't quite as perfect as the previous paragraph implies. There are things in the superclass structure that only make sense due to the subclass—such as methods that have been factored in such a way as to make it easier to override just the right kinds of behavior. So although most of the time I can modify the base class without having to understand subclasses, there are occasions where such mindful ignorance of the subclasses will lead me to breaking a subclass by modifying the superclass. However, if these occasions are not too common, the inheritance pays off—provided I have good tests to detect a subclass breakage.

So why would I want to change such a happy situation by using Replace Subclass with Delegate? Inheritance is a tool that can only be used once—so if I have another reason to use inheritance, and I think it will benefit me more than the premium booking subclass, I'll need to handle premium bookings a different way. Also, I may need to change from the default booking to the premium booking dynamically—i.e., support a method like `aBooking.bePremium()`. In some cases, I can avoid this by creating a whole new object (a common example is where an HTTP request loads new data from the server). But sometimes, I need to modify a data structure and not rebuild it from scratch, and it is difficult to just replace a single booking that's referred to from many different places. In such situations, it can be useful to allow a booking to switch from default to premium and back again.

When these needs crop up, I need to apply Replace Subclass with Delegate. I have clients call the constructors of the two classes to make the bookings:

*booking client*

```
aBooking = new Booking(show,date);
```

*premium client*

```
aBooking = new PremiumBooking(show, date, extras);
```

Removing subclasses will alter all of this, so I like to encapsulate the constructor calls with *Replace Constructor with Factory Function (334)*.

*top level…*

```
  function createBooking(show, date) {
    return new Booking(show, date);
  }
  function createPremiumBooking(show, date, extras) {
    return new PremiumBooking (show, date, extras);
  }
```

*booking client*

```
aBooking = createBooking(show, date);
```

*premium client*

```
aBooking = createPremiumBooking(show, date, extras);
```

I now make the new delegate class. Its constructor parameters are those parameters that are only used in the subclass, together with a back-reference to the booking object. I'll need this because several subclass methods require access to data stored in the superclass. Inheritance makes this easy to do, but with a delegate I need a back-reference.

*class PremiumBookingDelegate…*

```
constructor(hostBooking, extras) {
  this._host = hostBooking;
  this._extras = extras;
}
```

I now connect the new delegate to the booking object. I do this by modifying the factory function for premium bookings.

*top level…*

```
function createPremiumBooking(show, date, extras) {
  const result = new PremiumBooking (show, date, extras);
  result._bePremium(extras);
  return result;
}
```

*class Booking…*

```
 _bePremium(extras) {
   this._premiumDelegate = new PremiumBookingDelegate(this, extras);
 }
```

I use a leading underscore on `_bePremium` to indicate that it shouldn't be part of the public interface for `Booking`. Of course, if the point of doing this refactoring is to allow a booking to mutate to premium, it can be a public method.

Alternatively, I can do all the connections in the constructor for `Booking`. In order to do that, I need some way to signal to the constructor that we have a premium booking. That could be an extra parameter, or just the use of `extras` if I can be sure that it is always present when used with a premium booking. Here, I prefer the explicitness of doing this through the factory function.

With the structures set up, it's time to start moving the behavior. The first case I'll consider is the simple override of `hasTalkback`. Here's the existing code:

*class Booking…*

```
get hasTalkback() {
  return this._show.hasOwnProperty('talkback') && !this.isPeakDay;
}
```

*class PremiumBooking…*

```
get hasTalkback() {
  return this._show.hasOwnProperty('talkback');
}
```

I use *Move Function (198)* to move the subclass method to the delegate. To make it fit its home, I route any access to superclass data with a call to `_host`.

*class PremiumBookingDelegate…*

```
get hasTalkback() {
  return this._host._show.hasOwnProperty('talkback');
}
```

*class PremiumBooking…*

```
get hasTalkback() {
  return this._premiumDelegate.hasTalkback;
}
```

I test to ensure everything is working, then delete the subclass method:

*class PremiumBooking…*

```
~~get hasTalkback() {~~
  ~~return this._premiumDelegate.hasTalkback;~~
~~}~~
```

I run the tests at this point, expecting some to fail.

Now I finish the move by adding dispatch logic to the superclass method to use the delegate if it is present.

*class Booking…*

```
get hasTalkback() {
  return (this._premiumDelegate)
    ? this._premiumDelegate.hasTalkback
    : this._show.hasOwnProperty('talkback') && !this.isPeakDay;
}
```

The next case I'll look at is the base price.

*class Booking…*

```
get basePrice() {
  let result = this._show.price;
  if (this.isPeakDay) result += Math.round(result * 0.15);
  return result;
}
```

*class PremiumBooking…*

```
get basePrice() {
  return Math.round(super.basePrice + this._extras.premiumFee);
}
```

This is almost the same, but there is a wrinkle in the form of the pesky call on `super` (which is pretty common in these kinds of subclass extension cases). When I move the subclass code to the delegate, I'll need to call the parent case—but I can't just call `this._host._basePrice` without getting into an endless recursion.

I have a couple of options here. One is to apply *Extract Function (106)* on the base calculation to allow me to separate the dispatch logic from price calculation. (The rest of the move is as before.)

*class Booking…*

```
get basePrice() {
  return (this._premiumDelegate)
    ? this._premiumDelegate.basePrice
    : this._privateBasePrice;
}

get _privateBasePrice() {
  let result = this._show.price;
  if (this.isPeakDay) result += Math.round(result * 0.15);
  return result;
}
```

*class PremiumBookingDelegate…*

```
get basePrice() {
  return Math.round(this._host._privateBasePrice + this._extras.premiumFee);
}
```

Alternatively, I can recast the delegate's method as an extension of the base method.

*class Booking…*

```
get basePrice() {
  let result = this._show.price;
  if (this.isPeakDay) result += Math.round(result * 0.15);
  return (this._premiumDelegate)
    ? this._premiumDelegate.extendBasePrice(result)
    : result;
}
```

*class PremiumBookingDelegate…*

```
extendBasePrice(base) {
  return Math.round(base + this._extras.premiumFee);
}
```

Both work reasonably here; I have a slight preference for the latter as it's a bit smaller.

The last case is a method that only exists on the subclass.

*class PremiumBooking...*

```
get hasDinner() {
  return this._extras.hasOwnProperty('dinner') && !this.isPeakDay;
}
```

I move it from the subclass to the delegate:

*class PremiumBookingDelegate...*

```
get hasDinner() {
  return this._extras.hasOwnProperty('dinner') && !this._host.isPeakDay;
}
```

I then add dispatch logic to `Booking`:

*class Booking...*

```
  get hasDinner() {
    return (this._premiumDelegate)
      ? this._premiumDelegate.hasDinner
      : undefined;
  }
```

In JavaScript, accessing a property on an object where it isn't defined returns `undefined`, so I do that here. (Although my every instinct is to have it raise an error, which would be the case in other object-oriented dynamic languages I'm used to.)

Once I've moved all the behavior out of the subclass, I can change the factory method to return the superclass—and, once I've run tests to ensure all is well, delete the subclass.

*top level...*

```
function createPremiumBooking(show, date, extras) {
  const result = new ~~Premium~~Booking (show, date, extras);
  result._bePremium(extras);
  return result;
}

~~class PremiumBooking extends Booking ...~~
```

This is one of those refactorings where I don't feel that refactoring alone improves the code. Inheritance handles this situation very well, whereas using delegation involves adding dispatch logic, two-way references, and thus extra complexity. The refactoring may still be worthwhile, since the advantage of a mutable premium status, or a need to use inheritance for other purposes, may outweigh the disadvantage of losing inheritance.

## Example: Replacing a Hierarchy

The previous example showed using Replace Subclass with Delegate on a single subclass, but I can do the same thing with an entire hierarchy.

```
function createBird(data) {
  switch (data.type) {
    case 'EuropeanSwallow':
      return new EuropeanSwallow(data);
    case 'AfricanSwallow':
      return new AfricanSwallow(data);
    case 'NorweigianBlueParrot':
      return new NorwegianBlueParrot(data);
    default:
      return new Bird(data);
  }
}

class Bird {
  constructor(data) {
    this._name = data.name;
    this._plumage = data.plumage;
  }
  get name()    {return this._name;}

  get plumage() {
    return this._plumage || "average";
  }
  get airSpeedVelocity() {return null;}
}

class EuropeanSwallow extends Bird {
  get airSpeedVelocity() {return 35;}
}

class AfricanSwallow extends Bird {
  constructor(data) {
    super (data);
    this._numberOfCoconuts = data.numberOfCoconuts;
  }
  get airSpeedVelocity() {
    return 40 - 2 * this._numberOfCoconuts;
  }
}

class NorwegianBlueParrot extends Bird {
  constructor(data) {
    super (data);
    this._voltage = data.voltage;
    this._isNailed = data.isNailed;
  }

  get plumage() {
    if (this._voltage > 100) return "scorched";
    else return this._plumage || "beautiful";
  }
```

```
    get airSpeedVelocity() {
      return (this._isNailed) ? 0 : 10 + this._voltage / 10;
    }
  }
```

The system will shortly be making a big difference between birds tagged in the wild and those tagged in captivity. That difference could be modeled as two subclasses for `Bird`: `WildBird` and `CaptiveBird`. However, I can only use inheritance once, so if I want to use subclasses for wild versus captive, I'll have to remove them for the species.

When several subclasses are involved, I'll tackle them one at a time, starting with a simple one—in this case, `EuropeanSwallow`. I create an empty delegate class for the delegate.

```
class EuropeanSwallowDelegate {
}
```

I don't put in any data or back-reference parameters yet. For this example, I'll introduce them as I need them.

I need to decide where to handle the initialization of the delegate field. Here, since I have all the information in the single data argument to the constructor, I decide to do it in the constructor. Since there are several delegates I could add, I make a function to select the correct one based on the type code in the document.

*class Bird…*

```
  constructor(data) {
    this._name = data.name;
    this._plumage = data.plumage;
    this._speciesDelegate = this.selectSpeciesDelegate(data);
  }

  selectSpeciesDelegate(data) {
    switch(data.type) {
      case 'EuropeanSwallow':
        return new EuropeanSwallowDelegate();
      default: return null;
    }
  }
```

Now I have the structure set up, I can apply *Move Function (198)* to the European swallow's air speed velocity.

*class EuropeanSwallowDelegate…*

```
  get airSpeedVelocity() {return 35;}
```

*class EuropeanSwallow…*

```
  get airSpeedVelocity() {return this._speciesDelegate.airSpeedVelocity;}
```

I change airSpeedVelocity on the superclass to call a delegate, if present.

*class Bird…*

```
get airSpeedVelocity() {
  return this._speciesDelegate ? this._speciesDelegate.airSpeedVelocity : null;
}
```

I remove the subclass.

```
~~class EuropeanSwallow extends Bird {~~
  ~~get airSpeedVelocity() {return this._speciesDelegate.airSpeedVelocity;}~~
~~}~~
```

*top level…*

```
function createBird(data) {
  switch (data.type) {
    ~~case 'EuropeanSwallow':~~
      ~~return new EuropeanSwallow(data);~~
    case 'AfricanSwallow':
      return new AfricanSwallow(data);
    case 'NorweigianBlueParrot':
      return new NorwegianBlueParrot(data);
    default:
      return new Bird(data);
  }
}
```

Next I'll tackle the African swallow. I create a class; this time, the constructor needs the data document.

*class AfricanSwallowDelegate…*

```
  constructor(data) {
    this._numberOfCoconuts = data.numberOfCoconuts;
  }
```

*class Bird…*

```
selectSpeciesDelegate(data) {
  switch(data.type) {
    case 'EuropeanSwallow':
      return new EuropeanSwallowDelegate();
    case 'AfricanSwallow':
      return new AfricanSwallowDelegate(data);
    default: return null;
  }
}
```

I use *Move Function (198)* on airSpeedVelocity.

*class AfricanSwallowDelegate…*

```
  get airSpeedVelocity() {
    return 40 - 2 * this._numberOfCoconuts;
  }
```

*class AfricanSwallow…*

```
get airSpeedVelocity() {
  return this._speciesDelegate.airSpeedVelocity;
}
```

I can now remove the African swallow subclass.

```
~~class AfricanSwallow extends Bird {~~
  ~~// all of the body ...~~
  ~~}~~

function createBird(data) {
  switch (data.type) {
    ~~case 'AfricanSwallow':~~
      ~~return new AfricanSwallow(data);~~
    case 'NorweigianBlueParrot':
      return new NorwegianBlueParrot(data);
    default:
      return new Bird(data);
  }
}
```

Now for the Norwegian blue. Creating the class and moving the air speed velocity uses the same steps as before, so I'll just show the result.

*class Bird…*

```
selectSpeciesDelegate(data) {
  switch(data.type) {
    case 'EuropeanSwallow':
      return new EuropeanSwallowDelegate();
    case 'AfricanSwallow':
      return new AfricanSwallowDelegate(data);
    case 'NorweigianBlueParrot':
      return new NorwegianBlueParrotDelegate(data);
    default: return null;
  }
}
```

*class NorwegianBlueParrotDelegate…*

```
constructor(data) {
  this._voltage = data.voltage;
  this._isNailed = data.isNailed;
}
get airSpeedVelocity() {
  return (this._isNailed) ? 0 : 10 + this._voltage / 10;
}
```

All well and good, but the Norwegian blue overrides the `plumage` property, which I didn't have to deal with for the other cases. The initial *Move Function (198)* is simple enough, albeit with the need to modify the constructor to put in a back-reference to the bird.

*class NorwegianBlueParrot…*

```
get plumage() {
  return this._speciesDelegate.plumage;
}
```

*class NorwegianBlueParrotDelegate…*

```
  get plumage() {
    if (this._voltage > 100) return "scorched";
    else return this._bird._plumage || "beautiful";
  }

  constructor(data, bird) {
    this._bird = bird;
    this._voltage = data.voltage;
    this._isNailed = data.isNailed;
  }
```

*class Bird…*

```
selectSpeciesDelegate(data) {
  switch(data.type) {
    case 'EuropeanSwallow':
      return new EuropeanSwallowDelegate();
    case 'AfricanSwallow':
      return new AfricanSwallowDelegate(data);
    case 'NorweigianBlueParrot':
      return new NorwegianBlueParrotDelegate(data, this);
    default: return null;
  }
}
```

The tricky step is how to remove the subclass method for `plumage`. If I do

*class Bird…*

```
get plumage() {
  if (this._speciesDelegate)
    return this._speciesDelegate.plumage;
  else
    return this._plumage || "average";
}
```

then I'll get a bunch of errors because there is no plumage property on the other species' delegate classes.

I could use a more precise conditional:

*class Bird…*

```
get plumage() {
  if (this._speciesDelegate instanceof NorwegianBlueParrotDelegate)
    return this._speciesDelegate.plumage;
  else
    return this._plumage || "average";
}
```

But I hope that smells as much of decomposing parrot to you as it does to me. It's almost never a good idea to use an explicit class check like this.

Another option is to implement the default case on the other delegates.

*class Bird…*

```
get plumage() {
  if (this._speciesDelegate)
    return this._speciesDelegate.plumage;
  else
    return this._plumage || "average";
}
```

*class EuropeanSwallowDelegate…*

```
get plumage() {
  return this._bird._plumage || "average";
}
```

*class AfricanSwallowDelegate…*

```
get plumage() {
  return this._bird._plumage || "average";
}
```

But this duplicates the default method for plumage. And if that's not bad enough, I also get some bonus duplication in the constructors to assign the back-reference.

The solution to the duplication is, naturally, inheritance—I apply *Extract Superclass (375)* to the species delegates:

```
class SpeciesDelegate {
  constructor(data, bird) {
    this._bird = bird;
  }
  get plumage() {
    return this._bird._plumage || "average";
  }

class EuropeanSwallowDelegate extends SpeciesDelegate {
```

```
class AfricanSwallowDelegate extends SpeciesDelegate {
  constructor(data, bird) {
    super(data,bird);
    this._numberOfCoconuts = data.numberOfCoconuts;
  }

class NorwegianBlueParrotDelegate extends SpeciesDelegate {
  constructor(data, bird) {
    super(data, bird);
    this._voltage = data.voltage;
    this._isNailed = data.isNailed;
  }
```

Indeed, now I have a superclass, I can move any default behavior from `Bird` to `SpeciesDelegate` by ensuring there's always something in the `speciesDelegate` field.

*class Bird...*

```
selectSpeciesDelegate(data) {
  switch(data.type) {
    case 'EuropeanSwallow':
      return new EuropeanSwallowDelegate(data, this);
    case 'AfricanSwallow':
      return new AfricanSwallowDelegate(data, this);
    case 'NorweigianBlueParrot':
      return new NorwegianBlueParrotDelegate(data, this);
    default: return new SpeciesDelegate(data, this);
  }
}
// rest of bird's code...

get plumage() {return this._speciesDelegate.plumage;}

get airSpeedVelocity() {return this._speciesDelegate.airSpeedVelocity;}
```

*class SpeciesDelegate...*

```
  get airSpeedVelocity() {return null;}
```

I like this, as it simplifies the delegating methods on `Bird`. I can easily see which behavior is delegated to the species delegate and which stays behind.

Here's the final state of these classes:

```
function createBird(data) {
  return new Bird(data);
}
```

```
class Bird {
  constructor(data) {
    this._name = data.name;
    this._plumage = data.plumage;
    this._speciesDelegate = this.selectSpeciesDelegate(data);
  }
  get name()    {return this._name;}
  get plumage() {return this._speciesDelegate.plumage;}
  get airSpeedVelocity() {return this._speciesDelegate.airSpeedVelocity;}

  selectSpeciesDelegate(data) {
    switch(data.type) {
      case 'EuropeanSwallow':
        return new EuropeanSwallowDelegate(data, this);
      case 'AfricanSwallow':
        return new AfricanSwallowDelegate(data, this);
      case 'NorweigianBlueParrot':
        return new NorwegianBlueParrotDelegate(data, this);
      default: return new SpeciesDelegate(data, this);
    }
  }
  // rest of bird's code...
}

class SpeciesDelegate {
  constructor(data, bird) {
    this._bird = bird;
  }
  get plumage() {
    return this._bird._plumage || "average";
  }
  get airSpeedVelocity() {return null;}
}

class EuropeanSwallowDelegate extends SpeciesDelegate {
  get airSpeedVelocity() {return 35;}
}

class AfricanSwallowDelegate extends SpeciesDelegate {
  constructor(data, bird) {
    super(data,bird);
    this._numberOfCoconuts = data.numberOfCoconuts;
  }
  get airSpeedVelocity() {
    return 40 - 2 * this._numberOfCoconuts;
  }
}
```

```
class NorwegianBlueParrotDelegate extends SpeciesDelegate {
  constructor(data, bird) {
    super(data, bird);
    this._voltage = data.voltage;
    this._isNailed = data.isNailed;
  }
  get airSpeedVelocity() {
    return (this._isNailed) ? 0 : 10 + this._voltage / 10;
  }
  get plumage() {
    if (this._voltage > 100) return "scorched";
    else return this._bird._plumage || "beautiful";
  }
}
```

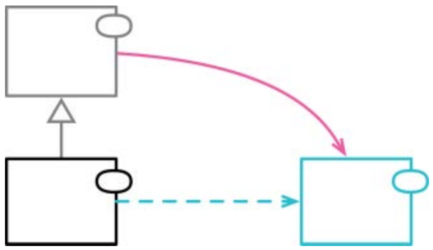
This example replaces the original subclasses with a delegate, but there is still a very similar inheritance structure in `SpeciesDelegate`. Have I gained anything from this refactoring, other than freeing up inheritance on `Bird`? The species inheritance is now more tightly scoped, covering just the data and functions that vary due to the species. Any code that's the same for all species remains on `Bird` and its future subclasses.

I could apply the same idea of creating a superclass delegate to the booking example earlier. This would allow me to replace those methods on `Booking` that have dispatch logic with simple calls to the delegate and letting its inheritance sort out the dispatch. However, it's nearly dinner time, so I'll leave that as an exercise for the reader.

These examples illustrate that the phrase "Favor object composition over class inheritance" might better be said as "Favor a judicious mixture of composition and inheritance over either alone"—but I fear that is not as catchy.

## Replace Superclass with Delegate

formerly: *Replace Inheritance with Delegation*



```
class List {...}
class Stack extends List {...}
```

⇓

```
class Stack {
  constructor() {
    this._storage = new List();
  }
}
class List {...}
```

### Motivation

In object-oriented programs, inheritance is a powerful and easily available way to reuse existing functionality. I inherit from some existing class, then override and add additional features. But subclassing can be done in a way that leads to confusion and complication.

One of the classic examples of mis-inheritance from the early days of objects was making a stack be a subclass of list. The idea that led to this was reusing of list's data storage and operations to manipulate it. While it's good to reuse, this inheritance had a problem: All the operations of the list were present on the interface of the stack, although most of them were not applicable to a stack. A better approach is to make the list into a field of the stack and delegate the necessary operations to it.

This is an example of one reason to use Replace Superclass with Delegate—if functions of the superclass don't make sense on the subclass, that's a sign that I shouldn't be using inheritance to use the superclass's functionality.

As well as using all the functions of the superclass, it should also be true that every instance of the subclass is an instance of the superclass and a valid object in all cases where we're using the superclass. If I have a car model class, with things like name and engine size, I might think I could reuse these features to represent a physical car, adding functions for VIN number and manufacturing date. This is a common, and often subtle, modeling mistake which I've called the type-instance homonym [mf-tih].

These are both examples of problems leading to confusion and errors—which can be easily avoided by replacing inheritance with delegation to a separate object. Using delegation makes it clear that it is a separate thing—one where only some of the functions carry over.

Even in cases where the subclass is reasonable modeling, I use Replace Superclass with Delegate because the relationship between a sub- and superclass is highly coupled, with the subclass easily broken by changes in the superclass. The downside is that I need to write a forwarding function for any function that is the same in the host and in the delegate—but, fortunately, even though such forwarding functions are boring to write, they are too simple to get wrong.

As a consequence of all this, some people advise avoiding inheritance entirely—but I don't agree with that. Provided the appropriate semantic conditions apply (every method on the supertype applies to the subtype, every instance of the subtype is an instance of the supertype), inheritance is a simple and effective mechanism. I can easily apply Replace Superclass with Delegate should the situation change and inheritance is no longer the best option. So my advice is to (mostly) use inheritance first, and apply Replace Superclass with Delegate when (and if) it becomes a problem.

## Mechanics

- Create a field in the subclass that refers to the superclass object. Initialize this delegate reference to a new instance.
- For each element of the superclass, create a forwarding function in the subclass that forwards to the delegate reference. Test after forwarding each consistent group.

  Most of the time you can test after each function that's forwarded, but, for example, get/set pairs can only be tested once both have been moved.
- When all superclass elements have been overridden with forwarders, remove the inheritance link.

## Example

I recently was consulting for an old town's library of ancient scrolls. They keep details of their scrolls in a catalog. Each scroll has an ID number and records its title and list of tags.

*class CatalogItem...*

```
  constructor(id, title, tags) {
    this._id = id;
    this._title = title;
    this._tags = tags;
  }

  get id() {return this._id;}
  get title() {return this._title;}
  hasTag(arg) {return this._tags.includes(arg);}
```

One of the things that scrolls need is regular cleaning. The code for that uses the catalog item and extends it with the data it needs for cleaning.

*class Scroll extends CatalogItem...*

```
  constructor(id,  title, tags, dateLastCleaned) {
    super(id, title, tags);
    this._lastCleaned = dateLastCleaned;
  }

  needsCleaning(targetDate) {
    const threshold =  this.hasTag("revered") ? 700 : 1500;
    return this.daysSinceLastCleaning(targetDate) > threshold ;
  }
  daysSinceLastCleaning(targetDate) {
    return this._lastCleaned.until(targetDate, ChronoUnit.DAYS);
  }
```

This is an example of a common modeling error. There is a difference between the physical scroll and the catalog item. The scroll describing the treatment for the greyscale disease may have several copies, but be just one item in the catalog.

It many situations, I can get away with an error like this. I can think of the title and tags as copies of data in the catalog. Should this data never change, I can get away with this representation. But if I need to update either, I must be careful to ensure that all copies of the same catalog item are updated correctly.

Even without this issue, I'd still want to change the relationship. Using catalog item as a superclass to scroll is likely to confuse programmers in the future, and is thus a poor model to work with.

I begin by creating a property in `Scroll` that refers to the catalog item, initializing it with a new instance.

*class Scroll extends CatalogItem…*

```
constructor(id,  title, tags, dateLastCleaned) {
  super(id, title, tags);
  this._catalogItem = new CatalogItem(id, title, tags);
  this._lastCleaned = dateLastCleaned;
}
```

I create forwarding methods for each element of the superclass that I use on the subclass.

*class Scroll…*

```
get id() {return this._catalogItem.id;}
get title() {return this._catalogItem.title;}
hasTag(aString) {return this._catalogItem.hasTag(aString);}
```

I remove the inheritance link to the catalog item.

```
class Scroll ~~extends CatalogItem~~{
  constructor(id,  title, tags, dateLastCleaned) {
    ~~super(id, title, tags);~~
    this._catalogItem = new CatalogItem(id, title, tags);
    this._lastCleaned = dateLastCleaned;
  }
```

Breaking the inheritance link finishes the basic Replace Superclass with Delegate refactoring, but there is something more I need to do in this case.

The refactoring shifts the role of the catalog item to that of a component of scroll; each scroll contains a unique instance of a catalog item. In many cases where I do this refactoring, this is enough. However, in this situation a better model is to link the greyscale catalog item to the six scrolls in the library that are copies of that writing. Doing this is, essentially, *Change Value to Reference (256)*.

There's a problem that I have to fix, however, before I use *Change Value to Reference (256)*. In the original inheritance structure, the scroll used the catalog item's ID field to store its ID. But if I treat the catalog item as a reference, it needs to use that ID for the catalog item ID rather than the scroll ID. This means I need to create an ID field on scroll and use that instead of one in catalog item. It's a sort-of move, sort-of split.

*class Scroll…*

```
constructor(id,  title, tags, dateLastCleaned) {
  this._id = id;
  this._catalogItem = new CatalogItem(null, title, tags);
  this._lastCleaned = dateLastCleaned;
}

get id() {return this._id;}
```

Creating a catalog item with a null ID would usually raise red flags and cause alarms to sound. But that's just temporary while I get things into shape. Once I've done that, the scrolls will refer to a shared catalog item with its proper ID.

Currently the scrolls are loaded as part of a load routine.

*load routine…*

```
const scrolls = aDocument
      .map(record => new Scroll(record.id,
                                record.catalogData.title,
                                record.catalogData.tags,
                                LocalDate.parse(record.lastCleaned)));
```

The first step in *Change Value to Reference (256)* is finding or creating a repository. I find there is a repository that I can easily import into the load routine. The repository supplies catalog items indexed by an ID. My next task is to see how to get that ID into the constructor of the scroll. Fortunately, it's present in the input data and was being ignored as it wasn't useful when using inheritance. With that sorted out, I can now use *Change Function Declaration (124)* to add both the catalog and the catalog item's ID to the constructor parameters.

*load routine…*

```
const scrolls = aDocument
      .map(record => new Scroll(record.id,
                                record.catalogData.title,
                                record.catalogData.tags,
                                LocalDate.parse(record.lastCleaned),
                                record.catalogData.id,
                                catalog));
```

*class Scroll…*

```
constructor(id,  title, tags, dateLastCleaned, catalogID, catalog) {
  this._id = id;
  this._catalogItem = new CatalogItem(null, title, tags);
  this._lastCleaned = dateLastCleaned;
}
```

I now modify the constructor to use the catalog ID to look up the catalog item and use it instead of creating a new one.

*class Scroll…*

```
constructor(id,  title, tags, dateLastCleaned, catalogID, catalog) {
  this._id = id;
  this._catalogItem = catalog.get(catalogID);
  this._lastCleaned = dateLastCleaned;
}
```

I no longer need the title and tags passed into the constructor, so I use *Change Function Declaration (124)* to remove them.

*load routine…*

```
const scrolls = aDocument
      .map(record => new Scroll(record.id,
                                ~~record.catalogData.title,~~
                                ~~record.catalogData.tags,~~
                                LocalDate.parse(record.lastCleaned),
                                record.catalogData.id,
                                catalog));
```

*class Scroll…*

```
constructor(id, ~~title, tags,~~ dateLastCleaned, catalogID, catalog) {
  this._id = id;
  this._catalogItem = catalog.get(catalogID);
  this._lastCleaned = dateLastCleaned;
}
```