The Addison-Wesley Signature Series

"Any fool can write code that a computer can understand. Good programmers write code that humans can understand."
—M. Fowler *(1999)*

A Martin Fowler Signature Book

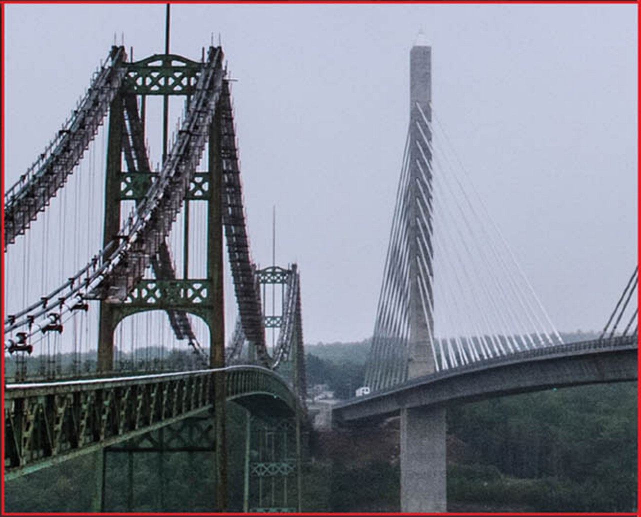
Refactoring

Improving the Design of Existing Code

Martin Fowler

with contributions by

Kent Beck



Second Edition

List of Refactorings

```
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`*.*

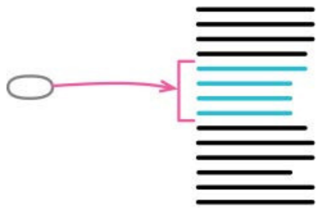
At this point you may be wondering, "What happens if more than one variable needs to be returned?"

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;
}
```

$\Downarrow$

```
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.