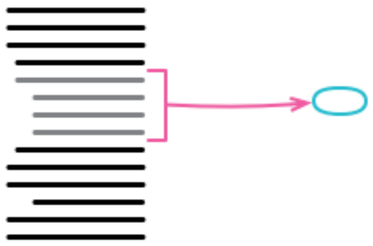
Extract Function

previous: A First Set of Refactorings

next: Inline Function



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

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

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

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

inverse of: Inline Function

formerly: Extract Method

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 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 or Replace Temp with Query 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 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—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, 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 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 and Split Variable.

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.

previous:
A First Set of Refactorings

next:
Inline Function